VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Virtuální prohlídka ve 3D

3D virtual tour

2015

Jan Orszulik

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Orszulik**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Virtuální prohlídka ve 3D
3D Virtual Tour

Zásady pro vypracování:

Cílem práce je navrhnout a implementovat aplikaci, která umožní management 3D scény s následnou vizualizací, renderování uživatelsky definovaných videí dle scénářů a export do zadaných formátů. Aplikace bude postavena nad vybraným enginem.
1. Přehled existujících nástrojů pro tvorbu a management 3D scén. Výběr vhodného engine a zdůvodnění tohoto výběru.
2. Návrh aplikace splňující požadovanou funkčost.
3. Implementace a výkonnostní testy.
4. Vyhodnocení testů.

Seznam doporučené odborné literatury:

Edward Angel: Interactive Computer Graphics, ISBN-10:032153586 (2009)

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2015

VYSOKÁ ŠKOLA BÁŇSKÁ-TECHNICKÁ UNIVERZITA * Fakulta elektrotechniky a informatiky * OSTRAVA *

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl.26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 6.května 2015 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6.května 2015 .....

Rád bych na tomto místě poděkoval vedoucímu práce Ing. Petru Gajdošovi, Ph.D., za odborné konzultace a poskytnutí zařízení pro výkonnostní testy. Své rodině za podporu během studia. Martinu Kuriplachovi za věcné připomínky, testování beta verze a poskytnutí hardwaru pro výkonnostní testy. Přítelkyni Janě Sklenářové za všeobecnou podporu.

Abstrakt

Cílem této diplomové práce bylo vytvořit 3D virtuální prohlídku areálu VŠB s vysokou mírou interaktivity ze strany koncového uživatele. Rovněž byly vytvořeny prostředky pro tvoření prezentačních materiálu jako výstup z aplikace. Výsledek obsahuje jak samotnou interaktivní aplikaci tak vizualizaci areálu VŠB. Součástí práce jsou výkonnostní testy na různých platformách. Projekt byl realizován v Unreal Engine 4.

Klíčová slova: 3D Virtuální prohlídka, Unreal Engine, prezentace, Vysoká škola báňská

Abstract

The goal of this master thesis is to create a 3D virtual tour of the VŠB areal with a high level of interactivity for the end user. It includes tools providing users with the ability to efficiently create presentation material as application output. This project consist of an interactive application and a visualization of the VŠB areal. Additionaly there are included, performance tests performed in multiple hardware environments. This project was realized in Unreal Engine 4.

Keywords: 3D Virutal tour, Unreal Engine, presentation, Vysoká škola báňská

Seznam použitých zkratek a symbolů

PBR	–	Physically based rendering
UE	–	Unreal Engine
OASIS	–	Organization For The Advancement Of Structured Information Systems
HTML	–	Hyper Text Markup Language
HLSL	–	High Level Shader Language
GLSL	–	OpenGL Shading Language
RT	–	Render thread
GT	–	Game thread

Obsah

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Virtuální prohlídky slouží k interaktivní prezentaci existujících nebo plánovaných objektů. Pojem samotný je opravdu široký, i panorama s možností otáčení lze považovat za virtuální prohlídku, nicméně dnešní informační doba a pokrok technologií umožňuje interaktivitu podstatně rozšířit a dát uživateli do rukou nástroj, který umožní lepší pochopení prostředí. To je požadováno jak z hlediska vytvoření pozitivní prezentace, tak možnosti usnadnit navigaci či předat obdobné informace předtím, než se daný uživatel do objektu skutečně vydá.

V případě požadavku na opravdu interaktivní prohlídku je možno přistupovat k projektu obdobně jako k počítačové hře. Cílem uživatele, tedy hráče, je získat informace, cílem vývojáře je poskytnout uživatelsky přívětivou, graficky a technologicky zvládnutou cestou dané informace.

K realizaci zadání, virtuální prohlídky, bylo rozhodnuto využití herního enginu, který obstará základní funkcionalitu jakožto renderování, zachycování vstupu ze strany uživatele, umožní rychlé vkládání nových grafických objektů a rychlé vývojové iterace za účelem dosažení požadované stylizace.

Prostředek na virtuální prohlídku může byt využit i samotným vývojářem, či jiným, nekoncovým uživatelem. V tomto případě může software, v případě, že má požadovaný výstup, sloužit k tvorbě statických prezentačních materiálu. Možnosti jako například vyznačení cesty a pořízení snímku obrazovky, jenž lze umístit na web, kde jej může shlédnout uživatel za využití minimálních prostředků, byly během této práce zhodnoceny a dle možností implementovány.

Volbu vhodného enginu ovlivňuje mnoho faktorů, které budou shrnuty a porovnány v následující kapitole. Patří mezi ně schopnost enginu vypořádat se s rozsáhlou scénou, kvalita a způsob renderování, platformy, na které je možné výsledný produkt nasadit a v neposlední řadě licenční podmínky.

2 Enginy

Enginy jsou nástroje pro vývojáře nabízející vestavěné funkce pro tvorbu interaktivních projektů, nejčastěji her na rozličné platformy. Nabízí prostředky pro renderování, správu assets, tedy modelů, textur, animací a zvukových efektů. Rovněž řeší hiearchii úrovní.

V současné době je na trhu k dispozici mnoho realtime enginů jež jsou dlouhodobě vyvíjeny špičkovými studii a za jistých licenčních podmínek poskytnuty menším vývojářským studiím a nezávislým vývojářům. Enginy se liší jak renderovací pipeline, nástroji a přístupem ke zdrojovému kódu, který umožňuje vývojářům provést jakoukoliv úpravu funkcionality.

2.1 Unity

Poprvé oznámen roku 2005, vyvíjen společností Unity Technologies [18]. Tento engine byl vyvíjen jako aplikačně všestranný a zaměřený na segment nezávislých vývojářů. Z předních enginů technologicky nejslabší, rozšířený mezi malými nezávislými studii, jen velmi vzácně použit pro profesionální tituly. Engine samotný je vytvořen v C++, scriptování probíhá běžně skrz vestavěné Mono v C# [22] či Javě, alternativně je možno využít Visual Studio. Právě díky podpoře C# si engine udržuje stabilní uživatelskou základnu. Popularitě rovněž pomáhá velký internetový sklad grafických, zvukových a jiných assets, tedy objektů, textur a materiálů. Vývoj materiálů je obtížný pro využití takzvaného shaderLab. Tvůrci Unity očekávají nákup materiálů na jejich skladu. V případě této práce nehrál sklad roli, jelikož veškeré potřebné objekty a materiály byly vytvořeny na míru projektu.

Jakožto silně multiplaformní engine umožňuje nasazení na Microsoft Windows, iOS, Android, BlackBerry, Windows Phone 8, Mac OS, Linux, PS3, PS4, Xbox One, WiiU, PS Vita a rovněž skrz web player.

Základní licenční verze unity je zdarma pro vývojáře s ročním ziskem do 100 000 USD. Unity pro je možno jednorázově licencovat za 1500 USD nebo za 75 USD měsíčně, avšak některé platformy, konkrétně android a iOS vyžadují vlastní licenci stejné ceny.

V současné době nejznámější projekt běžící na Unity je Hearthstone od Blizzard Entertainment, z nezávislých titulů pak Endless Space od Amplitude Studios.

2.2 Cryengine

Vyvíjen firmou Crytek [20]. Původně jako first person shooter engine pro hru Far Cry. Jedná se o velmi pokročilý engine a konkurenci Unreal Enginu, nabízí realtime dynamické osvětlení, deferred lightning rendrovací pipeline, pokročilé efekty jako screen space reflections a GPU tesselaci. Engine je vyvíjen v C++.

CryEngine lze nasadit na Microsoft windows, Playstation 3 a4, Xbox 360, Xbox One, Wii U, Android, iOS a Linux. Engine je poslední dobou kritizován za nedostatečnou podporu ze strany Cryteku a za citelně pomalejší vývoj v porovnání s Unreal Enginem a Unity.

Základní licence poskytované za 10 Euro měsíčně dostane vývojář Development kit, tedy engine samotný a vývojový editor. C++ a LUA api interface pro připojení vlastních rožšíření. Royalty fee jsou v případě této licence nulové. Znění licenčních podmínek ke zdrojovému kódu CryEnginu je krytý NDA (dohoda o mlčenlivosti).

Nejznámějšími příklady nasazení jsou zde hry série Crysis a série FarCry. Obě reprezentují schopnost enginu zvládat rozsáhlé otevřené úrovně. Tento fakt tvoří CryEngine vhodným kandidátem pro virtuální prohlídky rozsáhlých, realisticky zpracovaných ploch.

2.3 Ogre

Engine šíření pod licencí MIT [12], yvíjen Torus Knot Software Ltd [21] . Díky tomu je aktivně vyvíjen komunitou uživatelů. V komerčním prostředí se jedná o vzácně využívaný engine. Renderovací vlastnosti zaostávají za Unity, CryEngine a UnrealEnginem. Výhodou zůstává malá základní hardwarová náročnost.

Ogre lze nasadit na Microsoft Windows, Linux, Mac OSX, Android, iOS a Windows Phone.

Díky MIT [12] licenci se jedná o opravdový Open Source engine. Není nutno platit žádné měsíční poplatky ani poplatky ze zisku. Vývojář není povinen zveřejnit svůj zdrojový kód. V projektu je nutno uvést prohlášení o Open Source licenci enginu.

Jako příklad nasazení aplikace stojí za zmínku OpenMW, projekt, který rovněž řeší zpracování rozsáhlých ploch, avšak při nižší kvalitě zpracování než Unreal Engine či CryEngine.

2.4 Monogame

Open source engine vyvíjen MonoGame Team [10]. Vznikl roku 2009 jako náhrada XNA od Microsoftu. Původně zaměřen pouze na 2D hry, v dnešní době podporující moderní renderovací postupy včetně DX11. V současné době se jedná o open source. Vývoj zde není tak intuitívní jako v Unreal Enginu, Unity či CryEnginu, avšak nechává vývojáři úplnou kontrolu nad renderováním. Celkově se dá k monogame přistupovat spíše jako k souboru knihoven usnadňující vývoj díky již implementované, často používané, funkcionalitě. Oproti ostatním enginům se zaměřuje více na vývoj na konzole vydané firmou Microsoft, čímž pokračuje v zaměření XNA. Osobně tento engine z vlastní zkušenosti doporučuji vývojářům, kteří pracovali s XNA a plánují velmi specifický přístup k renderování neobvyklý pro ostatní enginy, například 2.5D hry či implementaci experimentálních přístupů. Dále je výborný pro modernizaci existujících XNA projektů, jelikož používá stejné api.

Z 3D aplikací je vhodné zmínit Infinite Flight, což je simulátor letadel běžící na mobilních zařízeních. MonoGame však nebyl komerčně využit pro projekt zaměřený na rozsáhlé projekty vysoké kvality.

Monogame lze nasadit na Microsoft Windows, Linux, Mac OSX, Android, Xbox 360 a Xbox.

Jedná se o úplný opensource software s veřejným přístupem přes GitHub.

2.5 Unreal Engine 4

Nejmodernější a nejrychleji vyvíjející se engine. Jeho první předchůdce vznikl již v roce 1998, naprogramován firmou Epic Games [19]. Původně určen pro žánr first person shooter, tedy střílečky z pohledu vlastních očí, engine se rychle rozšířil a jeho úpravy byly využity napříč všemi herními žánry. Aktuální verze, Unreal Engine 4.0, byla vydána v květnu 2014.

V rámci velmi dostupné licence dostane uživatel přístup ke kompletnímu zdrojovému kódu, směr vývoje do jisté míry díky tomu určuje komunita. Renderovací pipeline je kompletně deferred 3.1 s translucentními materiály a možností vlastních G-bufferů. Engine je implementován v C++. V rámci scriptování poté nabízí takzvané Blueprinty, jedná se o druh grafického programování a hlavně debuggování s vizualizací datových toků za běhu programu. V rámci vývoje je možno provést změnu kódu (například skrz Visual Studio) přímo během testování bez nutnosti vypínat engine a ihned vidět výsledky, tato

funkce se nazývá Hot Reload. Tato možnost značně zrychluje iteraci vývoje a ladění výsledného produktu.

Unreal Engine 4.0 lze nasadit na Microsoft Windows, Linux, Mac OS, Android a iOS, Playstation 4 a Xbox One, do budoucna je plánována podpora Windows phone. Rovněž je možné projekt prezentovat skrz webový prohlížeč za využití HTML5 a OpenGL.

Licence byla po vypuštění poskytována formou předplatného s cenou 20 euro za měsíc v rámci Evropy. V jejím rámci byly poskytovány pravidelné updaty a rozšíření. Od 2.3.2015 byl zrušen poplatek a používání Unreal Enginu 4 je zdarma. Stále však platí Royalty fee, tedy poplatek ze zisku, činí 5% z ročního zisku nad 3000 USD.

Vzhledem k datu vydání enginu není v současné době jednoduché uvést dokončený projekt, jenž by dobře prezentoval schopnosti enginu. Jako open source projekt ve vývoji je vhodné zmínit Unreal Tournament. Z dalších připravovaných titulů potom DreadNought a Eve: Valkyrie, který se zaměřuje na experimenty v oblasti virtuální reality.

V současné době má kdokoliv volný přístup ke zdrojovému kódu Unreal Enginu. Díky tomu se přímo nabízí jako výborný zdroj informací o špičkových řešeních v oblasti realtime grafiky.

Updaty jsou distribuovány skrze spouštěcí aplikaci, takzvaný Launcher. Po přihlášení získá registrovaný uživatel přehled o aktuální verzi, může spravovat nainstalované instance enginu. Rovněž má přístup ke všem svým existujícím projektům. Dále Launcher nabízí přístup k marketplace, tedy místu sloužícímu pro nákup a prodej modelů, materiálů, zvukových efektů, blueprintů, či celých projektů.

Samozřejmě renderování není jediná schopnost enginu, mnoho lidí by řeklo, že ani hlavní. Stejně tak zdrojový kód obsahuje síťové řešení, optimalizaci vstupu uživatele, multithreading, doslova vše co je třeba k vybudování špičkové hry či projektu založené na stejné technologii.

Přístup ke zdrojovému kódu je však přínosný i pro autory Unreal Enginu. Komunita se aktivně podílí na vývoji, přidávání nových možností a vylepšování stávajících. Díky tomu se Unreal Engine stal nejrychleji vyvíjejícím se herním enginem současnosti a nutno podotknout, že již na trh vešel ve své době jako nejmodernější.

Zdroj UE je napsán v C++ jak již bylo dříve zmíněno, je možno použít libovolné vývojové prostředí.

Přístup ke zdrojovému kódu probíhá přes GitHub. Pozvání uživatele proběhne po registraci učtu, bližší informace jsou k dispozici na oficiální webové stránce [8].

Epic využívá platformy AnswerHub [9] pro řešení dotazů uživatelů, feedback a hlášení chyb. Využívá systém reputace pro hodnocení příspěvků uživatelů a jedná se o moderovaný systém.

2.6 Volba enginu pro realizaci práce

Pro realizaci tohoto projektu jsem zvolil Unreal Engine 4.0. Důvody jsou použité technologie, efektivní renderovací pipelne, pokročilý navigační mesh, silná základní AI, jenž je nabízí potencionální využití při hledání cest, způsob realizace shaderu, kompletní přístup ke zdrojovému kódu a rychlý vývoj díky hot build, tedy možnosti editace zdrojového kódu za běhu enginu a výhodné licenční podmínky.

Unreal Engine umožňuje efektivně spravovat rozsáhlé scény, aktualizace objektů je jednoduchá a v případě změny reálného objektu, který je cílem virtuální prohlídky, je triviální provést adekvátní změnu ve scéně a předávat tak uživateli aktuální informace. Dále je možno v rámci OpenGL nasadit projekt na web a zpřístupnit jej přes webový prohlížeč.

3 Renderování v reálném čase

Tato kapitola se bude zabývat řešením renderování samotného, tedy způsobu převodu informací v podobě modelů, textur, fontů a shaderu do výsledné rastrové podoby.

Vzhledem k vytvoření možnosti spojitého procházení areálu je nutné renderování v reálném čase. Všechny zmíněné herní enginy jsou tohoto schopny, předpoklad je hardware schopný renderovat danou scénu při dostatečné snímkové frekvenci vytvářející iluzi plynulého pohybu.

Efektivní renderování v reálném čase [23] je umožněno díky pro to určeným grafickým kartám, gpu. Renderování v reálném čase je rychle vyvíjející se oblast. Předešlý takzvaný forward rendering [2] byl nahrazen deferred renderingem, avšak trend se nyní upíná k novým alternativám kvůli omezením, jenž deferred rendering přinesl.

3.1 Deferred rendering

Jedná se o technologicky pokročilejší alternativu forward renderingu [2]. Základní myšlenka forward renderingu byla postupné kompletní renderování jednotlivých objektů. V rámci jednoho průchodu tedy byla vyrenderována geometrie, aplikovány textury a vypočítáno světlo, po dokončení objektu se grafická karta přesunula na následující objekt, kde byl opět proveden kompletní výpočet. Deferred rendering se označuje jako screen space technika, což znamená, že veškerá data přímo relevantní k výslednému osvětlení jsou zpracovány v rámci rozlišení a rozsahu výsledné obrazovky, osvětlení se tedy renderuje pro všechny viditelné objekty zároveň. Postupně jsou vyrendrovány potřebné průchody, základní:

- **Diffuse Buffer** - difuzní buffer obsahuje informace o barvě objektů, ať už základní barva získaná na základě obarvení vertexů nebo textur.
- **Emission / Self illumination Buffer**- emisní buffer obsahuje informace, jenž jsou renderovány additivně k výsledku po vypočtení osvětlení. V tomto projektu využito například pro zvýraznění parkovacích míst.
- **Depth Buffer** - hloubkový buffer zachovává hloubku scény s vysokou přesností, běžně je použit jeden kanál o 24 až 32 bitové hloubce.
- **Normal Buffer** - normálový buffer obsahuje informace o normálách objektu, tedy orientaci normál přepočítaných a renderovaných ve World space.

Následně je v kombinaci s informací o světlech spočítán výsledný obraz.

Ve forward renderingu bylo potřeba pro každý objekt počítat světlo individuálně, tedy

$$O(F_n L_n) \tag{1}$$

kde F_n je počet geometrických fragmentů a L_n počet světel. Zároveň je hodně výpočtů zbytečných například kvůli překrytí objektů.

Naproti tomu náročnost Deferred renderingu

$$O(Screen_r L_n) \tag{2}$$

kde $Screen_r$ je rozlišení obrazovky, tedy počet viditelných pixelů, pro které je třeba spočítat osvětlení.

Nevýhodou deferred renderingu je složitost řešení průhledných objektů. Ty je možno řešit v rámci omezených možností ve speciálních bufferech, je tedy nutno je rendrovat zvlášť a rovněž pro ně počítat zvlášť stínování. V důsledku toho není možno provádět antialiasing během renderování geometrie, jelikož by i na neprůhledných objektech způsobila částečně průhledné fragmenty v oblasti hran. Je tedy řešen v rámci postprocessu, v případě Unreal Enginu je využit algoritmus FXAA [16].

Obrázek 1: Ukázka hlavních bufferu deferred renderingu, difuzní, hloubkový a normálový

https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/HowTo/Transparency/inde

3.2 Fyzikálně založené renderování

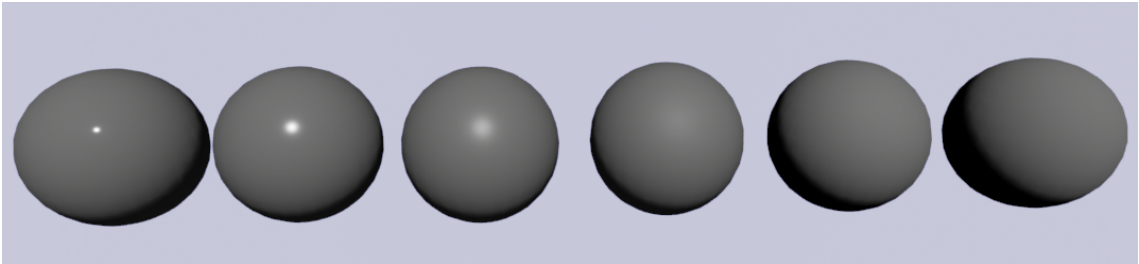
Jedná se o poslední dobou rozšiřující se přístup k renderování a materiálům v real time grafice. Cílem PBR [11] je přiblížení se k chování světla v reálném světě a vytvoření více univerzálních materiálů, které mají stabilní kvalitu napříč různým prostředním.

PBR chápe každý materiál jako reflexivní, specularity tedy není dodatečný efekt materiálu, ale je přímo vázán k ostatním atributům. Konkrétně specularity a specularity power jsou nahrazeny jedním parametrem zvaným roughness, tedy drsnost povrchu. Ten určuje drsnost povrchu, při nižší hodně se materiál stává více reflexivním, odráží jak silné zdroje světla, tak své okolí. Díky využití zákonu o zachování energie je možno dosáhnout reálných odrazů světla za použití jediného parametru, ať už je ním konstanta či textura v odstínech šedi.

Jednou z implementací odrazů světla v PBR je Cook - Torrance [17]. Jedná se o takzvanou biderectional reflectance distribution function, tedy funkce s obousměrnou distribucí odrazivosti, alternativy této funkce jsou více známé Blinn či Phong stínování, které ale nesplňují podmínky pro PBR. Všeobecně jakýkoliv model, který vyžaduje pro výpočet odlesku světla, specularity, dva parametry, porušuje zákon zachování energie. Na obrázku 2 lze vidět ovládání odlesku pomocí jednoho parametru, intenzita celkového odlesku snížena úměrně ploše odlesku.

Pro vypočítání Cook - Torrance stínování potřebujeme vypočítat tři termy: fresnel, geometrický a drsnost. V příloze A je k dispozici má implementace Cook - Torrance s komentáři . Přesto že se nejedná přímo o implementaci použitou v Unreal Enginu může posloužit pro lepší pochopení fyzikálně založeného stínování.

Stejně jako většina běžně používaných modelů je možno i tento akumulovat a použít jej tedy v deferred renderingu.



Obrázek 2: Ukázka chování odrazu světla v PBR, roughness z leva doprava od 0.0 do 1.0 odstupňováno po 0.2 krocích

3.3 Postprocessing

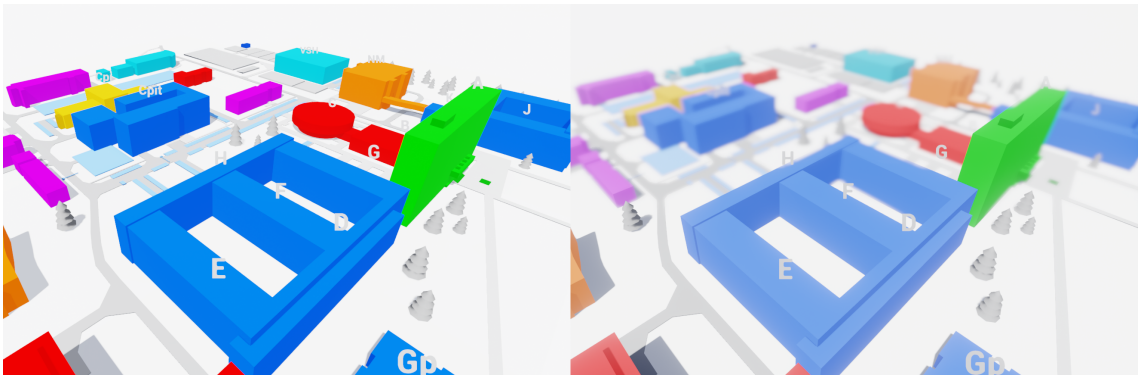
Postprocessing, jak název napovídá, probíhá na konci renderovacího cyklu. Umožňuje ovlivňovat celkový vzhled a ladit atributy jako kontrast scény, saturace, globální illumi-

naci, 3D LUT gradient či přidat další efekty jako ambient occlusion, antialliasing, screenspace reflections. K realizaci využívá přístupu k offscreen bufferům, zejména depth bufferu a scene bufferu. Scene buffer obsahuje data, která by byla bez postprocessingu přímo rendrována na obrazovku.

V rámci stylizace projektu bylo vytvořeno několik různých postprocessů, které jsou mezi sebou přepínány dle zvolené stylizace. Nejzajímavější z nich je postprocess využit pro "cartoon"vzhled. Využívá vlastního shaderu, který je dále rozebrán v kapitole Materiály 3.4.

Na obrázku 3 je možno vidět efekt různých postprocesu na stejné scéně. Obrázek vpravo obsahuje:

- Depth of field - simulace hloubky ostrosti za využití hloubkového bufferu
- Saturation - snížení sytosti barev
- Shadow crush - potlačení stínu a tmavých míst



Obrázek 3: Ukázka dvou různých post procesů na stejné scéně

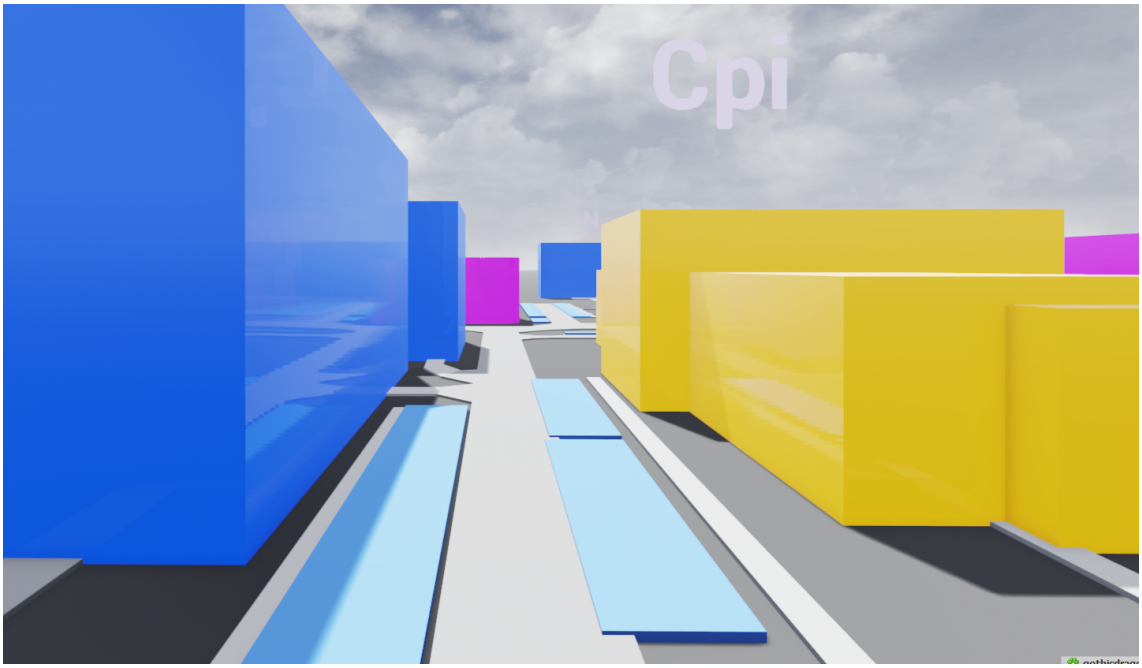
3.3.1 Screen space reflections

Obrazy v prostoru obrazovky jsou moderní prvek nejnovější generace realtime enginu [4]. Zvyšují realitu prostředí a objektu, popřípadě umožňují rozšířit stylizaci, jak je tomu v případě tohoto projektu.

Jedná se o součást postprocessu, pokud tedy máme potřebné grafické buffery, čímž je depth buffer, normal buffer a buffer obsahující informace o míře odrazivosti povrchu, tedy v PBR roughness, můžeme spočítat odrazy.

Realizace probíhá následovně: Je spočítán vektor odrazu pro každý renderovaný pixel za využití depth bufferu a normal bufferu. Podél daného vektoru je vytvořen racast. Jsou vytvořeny vzorky a je ověřeno, jestli je paprsek stále v rámci hloubky scény. Pokud ano, je získán vzorek barvy.

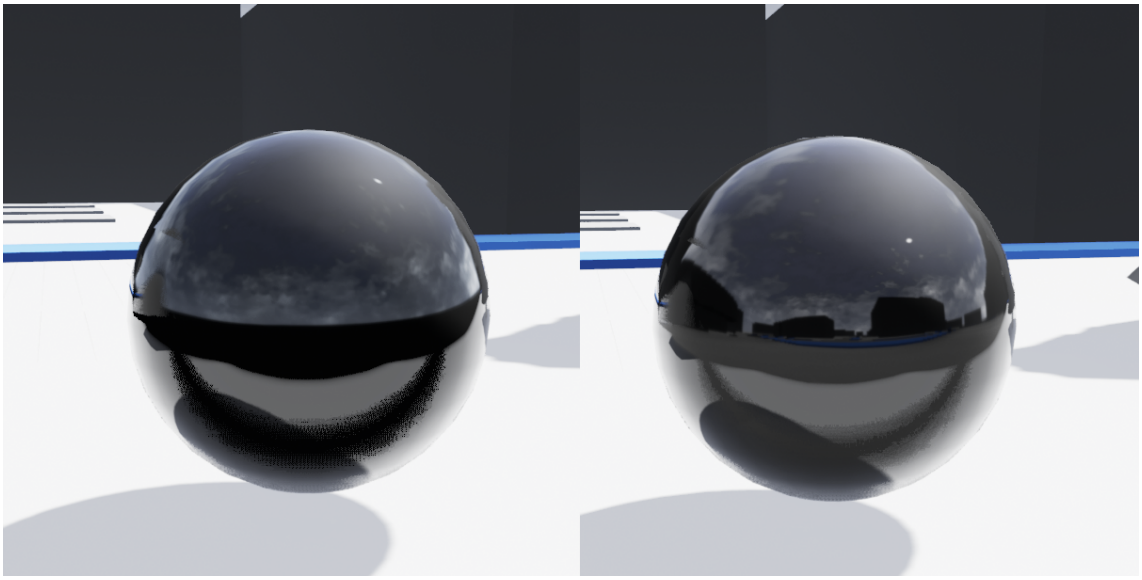
Samozřejmě jsou zde jistá omezení, samotný fakt, že se jedná o odrazy v prostoru již vyrenderovaného obrazu znamená, že není možno za normálních okolností odrážet objekty mimo něj. Jako řešení se zde nabízí takzvané Reflection Capture. Na předem zadaném místě je uložena informace, která obsahuje nasnímaný okolní prostor. Tato informace je statická, nezahrnuje tedy dynamický pohyb, přesto může značně pomoc výsledku jak je možno vidět na obrázku 5.



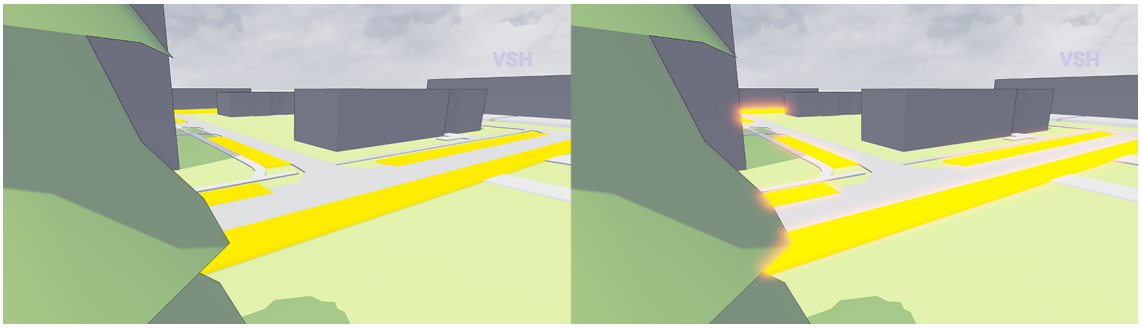
Obrázek 4: Screenspace odrazy

3.3.2 Bloom

Zajímavým i praktickým efektem je bloom. Jedná se o simulaci přírodního jevu při pohledu na velmi světlý objekt. V tomto projektu je bloomu využito při stylizaci Cartoonu a je velmi viditelný pří zvýraznění parkovacích míst. Vytváří září v okolí objektu, dále zdůrazňující daný objekt.



Obrázek 5: Vlevo odraz prostředí na kouli bez použití Reflection Capture, vpravo s ní



Obrázek 6: Porovnání zvýraznění parkovacích míst bez využití bloom efektu a s ním

Obecně efekt samotný probíhá v realtime grafice následovně:

- Na bufferu výsledné scény je aplikován shader, jenž izoluje části nad určitou světlost. Výsledek je uložen do dočasného bufferu.
- Na výsledku je provedeno dvou průchodové Gaussovo rozostření.
- Výsledek předchozího kroku je rendrován aditivně3.1 na buffer scény.

Definice 3.1 *Aditivní renderování je přičtení barevné hodnoty daného fragmentu k hodnotě již existujícího fragmentu. Pokud například máme původní hodnotu fragmentu v* $Color_{RGB} =$

$(50, 10, 0)$ *a na něj aditivně renderujeme* $Color_{RGB} = (0, 60, 10)$*, výsledná hodnota fragmentu bude* $Color_{RGB} = (50, 70, 10)$.

Gaussovo rozostření je relativně drahý proces, který vyžaduje velký počet vzorků textury, či v tomto konkretním případě bufferu.

Obecně se Gaussovo rozostření řeší pomocí konvoluce s jádrem využívajícím všeobecné Gaussovy rovnice, pro jenž 2D vypadá následovně:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

Kde x je horizontální vzdálenost daného pixelu od původního pixelu a y je vzdálenost vertikální, σ je standardní odchylka Gaussovy funkce. Výsledkem je vážený průměr okolních pixelů v každém bodě.

Vzhledem k tomu, že se jedná o filtrování 2D obrazu s diskrétními body, tedy pixely, je možné jádro konvoluce definovat pomocí 2D matice.

Následně je využito separabilnosti jádra 3.2. Díky tomu se dá 2D konvoluce rozložit na dva průchody 1D konvolucí, vertikální a horizontální, což sníží výpočetní složitost z $O(MNmn)$ na $O(MN(m + n))$.

Definice 3.2 *Jádro filtru je separabilní pokud* $h(x, y) = h_1(x) \cdot h_2(y) = \delta * h_1(x, \cdot) * h_2(\cdot, y)$*, kde* δ *je Diracova funkce.*

Vzhledem k časové náročnosti je třeba tento proces řešit na GPU.

- Na CPU jsou předpočítány offsety a váhy dle požadované míry rozostření, tvořící 2D matici.
- Offsety, váhy a požadovaná textura, či buffer jsou poslány shaderu zajišťující horizontální průchod 2.
- Na základě offsetu a vah dochází k 1D konvoluci.
- Výsledek je obdobně zpracován shaderem zajišťujícím vertikální průchod.

```
for (int i=1; i<sampleCount; i++) {
    outputTexture +=
        tex2D(TextureSampler, ( PSIn.TexCoord+float2(offset[i], 0.0) /viewport.x ) )
            * weight[i ];
    outputTexture +=
```

```
            tex2D(TextureSampler, ( PSIn.TexCoord-float2(offset[i], 0.0) /viewport.x ) )
                * weight[i ];
        }
```

Výpis 1: Kód pro výpočet horizontálního průchodu Gaussova rozostření v HLSL

Díky tomu, že probíhá na GPU, je možno využít redukci počtu vzorků díky lineárnímu vzorkování textury [13].

Pokud při vzorkování textury na GPU nepoužíváme přímo střed pixelu, je vzorkem průměr z okolních pixelů v závislosti na vzdálenosti jejich středu od námi vzorkovaného bodu, v případě dvouprůchodového Gaussova rozostření, kdy každý průchod je jednorozměrný, je pro nás relevantní hodnota mezi dvěma pixely. Tento přístup tedy snižuje počet vzorků na polovinu při zachování stejného množství informací.

Pro využití tohoto faktu je třeba přepočítat jádro filtru pro Gaussovo rozmazání. Váhy jsou vypočítány dle vztahu

$$weight_L(t_1, t_2) = weight_D(t_1) + weight_D(t_2) \tag{4}$$

Offsety pak

$$offset_L(t_1, t_2) = \frac{offset_D(t_1) * weight_D(t_1) + offset_D(t_2) * weight_D(t_2)}{weight_L(t_1, t_2)} \tag{5}$$

Univerzální implementaci obsahující HLSL shader a vypočítání lineárních offsetů a vah založené na článku [13] můžete najít v příloze A.

3.4 Materiály

Materiál je v terminologii enginu struktura shaderu kompletní tak, aby v rámci daného Enginu umožnila vyrenderování objektu. Moderní shader má několik částí:

- **Vertex shader** - Je proveden na vertexech vstupní geometrie. Určuje transformuje pozici na základně projekčních matic. Dále umožňuje transformovat na základě libovolné matematické funkce, jenž může být za běhu ovládána parametrem, v tomto projektu využito pro materiál budov3.4.4.
- **Geometry shader** - Volitelná část shaderu, jenž existuje od DirectX 10. Umožňuje přidávat geometrii, tedy vertexy do již existujících objektů. Vhodné například při tvoření 2D spritu. V moderní grafice často využíváno pro GPU částice, jelikož umožňuje dále snížit využití CPU a tvořit část geometrie na GPU.

- **Pixel(Fragment) shader - Uroveň shaderu** starající se o rasterizaci, pracuje s pixely renderované scény, tedy kompletně 2D. V moderní grafice je zde provedeno počítání světla a dalších efektů. Základní funkčnost je například aplikace textur.
- **Hull shader / Domain shader / Tessellation control shader** - podobně jak Geometry shader slouží k přidávání geometrie. Jsou optimalizované na přidání velkého počtu vertexů díky hardwarové struktuře GPU. Slouží k přidávání detailů, jež není možné řešit přes normálové mapy v Pixel shaderu.
- **Compute shader** - Slouží k provedení všeobecných algoritmů na GPU a tedy přesunutí části zátěže z CPU na GPU. Využito například při takzvaném Box Blur, což je alternativa ke Gaussovu rozostření.

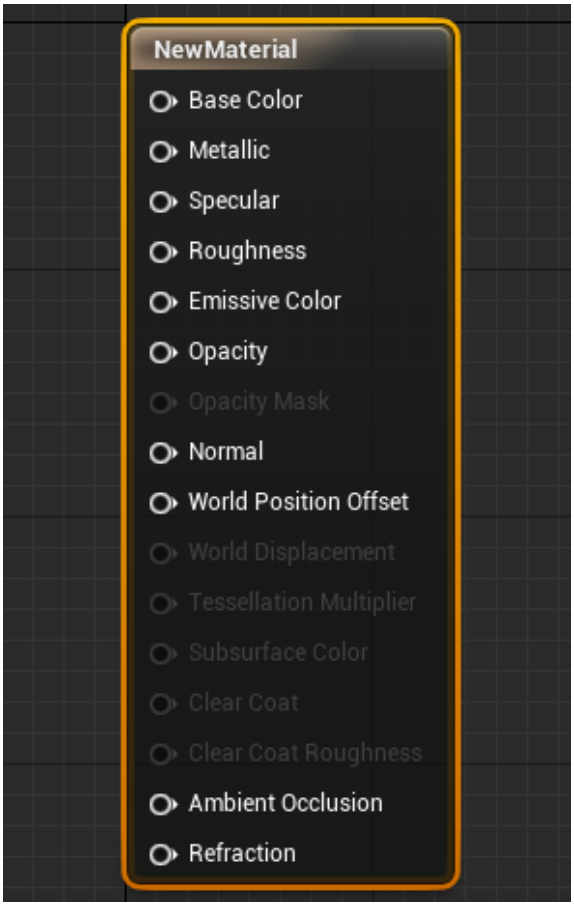
Vzhledem k deferred renderingu je aplikováno na jeden objekt několik shaderů. Nejdříve shadery v rámci materiálu přiřazenému konkrétnímu objektu. Následně pak v rámci screen space jsou na celou obrazovku aplikovány shadery řešící stínování objektů. Následně pak volitelně shadery pro post procesing.

V UE jsou materiály řešeny několika způsoby. Hlavní vývojovým nástrojem je zde material editor, využívající uzly (nody) k vytvoření materiálu. Tyto materiály je možno zkompilovat jako HLSL [24] či GLSL [25]. Tento krok je řešen automaticky při buildu projektu. Uzly reprezentují jednotlivé funkce shaderu, jež tvoří materiál, využívají stejnou syntaxi jako HLSL a jsou kompatibilní. Pomocí tohoto souboru je rovněž možno přidávat další nové uzly a rozšiřovat tím možnosti enginu, nové uzly je možno tvořit rovněž v rámci material editoru samotného, v tom případě se běžně nazývají makry. Faktem zůstává, že materiál vytvořený v node editoru je naprosto roven materiálu napsaném ručně co se týče výkonnosti. Naopak co se týče uživatelské přívětivosti, umožňuje node editor real time preview, tedy náhled v reálném čase, výsledného materiálu a jednotlivých nodů.

3.4.1 Výstup shaderu

V následující části budou stručně shrnuty výstupy materiálu použity v rámci toho projektu, kompletní přehled je uveden v oficiální dokumentaci.

- **Color Basic** - Výstup do prvního renderovacího bufferu, zpravidla difuzní, neboli albedo, určuje barvu a intenzitu daného pixelu před aplikací světla.



Obrázek 7: Výstupu shaderu

- **Metallic** - Určuje kovový vzhled povrchu, konkrétně míru aplikací vlastní barvy v porovnání s prostředím, tedy například vliv odrazu. Vstup v rozsahu 0 až 1, pro čisté povrchy se používají pouze hodnoty 0 až 1 dle daného typu, pro realistickou simulaci kombinovaného povrchu, tedy například zaprášeného, se využívají hodnoty mezí. Tento projekt využívá metallic parametr u některých shaderů, ne za účelem reálnosti, ale za účelem dosažení požadovaného vzhledu konkretní stylizace.

- **Roughness** - Určuje drsnost povrchu z pohledu fyzikálně založeného renderování světla3.2. Nahrazuje specularity a glossiness mapu.

- **Emission** - Rendrováno do bufferu, jenž je aplikován aditivně k pixelu s již vypočteným světlem. Tato informace samotná je tedy neovlivněna okolním světlem

a je vhodná pro simulaci objektů emitujících světlo v rámci materiálu, za normálních okolností neovlivňuje okolní objekty jako samostatný zdroj světla, nicméně existující jisté technologie které toto umožňují, byť nepracují dynamicky, nejsou renderovány realtimově, ale předpočítány a uloženy do textur.

- **World position offset** - Oproti předcházejícím se jedná o operaci ve Vertex Shaderu. Umožňuje modifikovat vertexy v závislosti na informaci materiálu, běžně se používá v kombinaci s GPU tesselaci objektu, v tomto projektu byl využit k automatickému otáčení textu a snižování budov.

3.4.2 Vytváření shaderu

Přesto, že realizace probíhá pomocí uzlů, každý uzel představuje funkci, jenž lze reprezentovat například HLSL či GLSL kódem.

Uzly se dělí na tři typy.

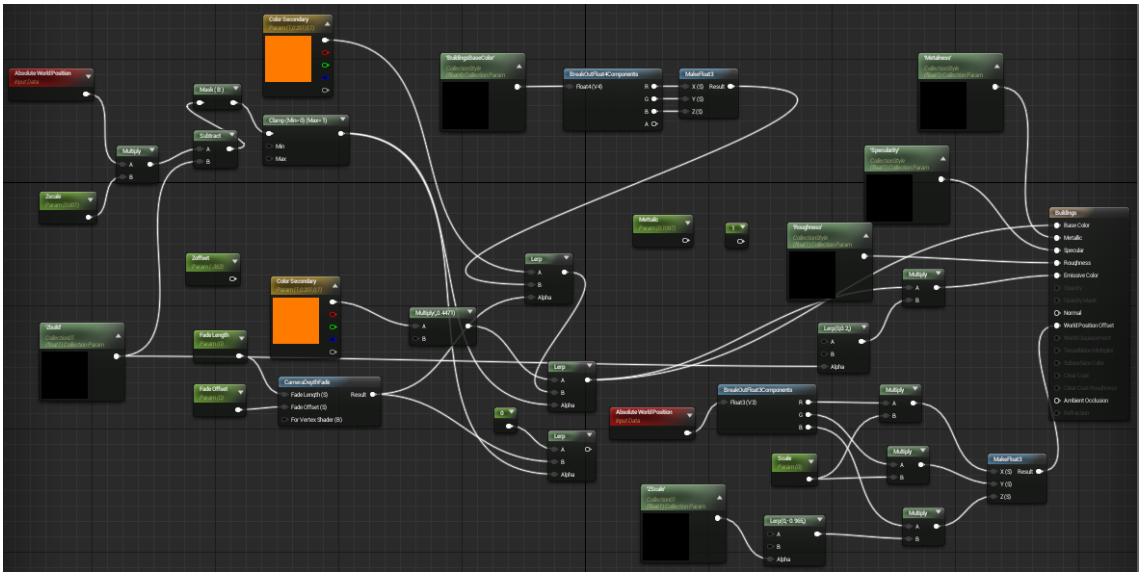
- Vstupní - vytváření vstupní informace pro daný shader, může se jednat o konstanty či parametry.
- Funkce - uzly, které zpracují vstup a poskytují výstup pro další zpracování
- Výstupní - méně používané uzly, které směrují výstup do speciálních, uživatelem vytvořených grafických bufferů, umožňují tak předávání informací mezi jednotlivými shadery.

Postupným přidáváním uzlů je tedy budován shader, jenž je možno následně zkompilovat pro požadovaný jazyk jako například HLSL či OpenGL.

Přímo v editoru je rovněž možno zobrazit HLSL kód daného shaderu. Možnost komentovat části shaderu je samozřejmostí usnadňující dokumentaci a znovupoužitelnost v budoucích projektech.

3.4.3 Parmetrizace shaderu

Pro demonstraci shaderu budou rozebrány tři materiály vytvořeny pro tento projekt, materiál použitý na budovy, jenž zahrnuje parametrizaci s propojením na interaktivní blueprint uživatelského rozhraní, material collection, PBR vlastnosti, world position offset a



Obrázek 8: Ukázka materiálu použitého na budovách

využití depth bufferu. Dále materiál využit ve stylizaci pojmenované "Cartoon"pro vytvoření cartoon hran objektu a sjednocení barev. Jako poslední příklad materiál stopy, který je využit v jednoduchém částicovém systému.

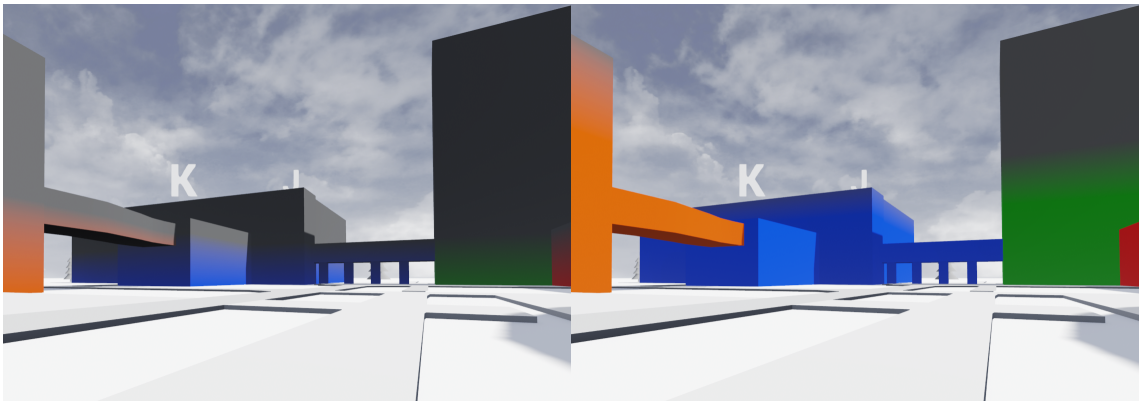
Projekt měl požadavky v rámci vizuální informace, které bylo nutné řešit skrz využití materiálu. Jedním z nich byla informace o budovách, základní vzhled měl byt jednoduchý v málem rozsahu barev, rovněž však je třeba zobrazit barevné rozlišení budov na vyžádání uživatele. Pro tento prvek bylo využité lineární interpolace mezi dvěma barvami v závislosti na vstupu dynamického parametru.

3.4.4 Materiál budov

Stěžejním materiálem je materiál budov. Budovy samotné jsou dominantou areálu a mají hlavní vliv na výsledek virtuální.

Obarvení na základě vzdálenosti probíhá následovně. Dle informací získaných z Depth Bufferu je zjištěna hloubka pixelu, tedy vzdálenost od kamery samotné. Pomocí funkce Depth Fade je vybudován gradient za využití dynamických parametrů. Funkce Depth Fade pracuje následovně:

1. Je vypočítán vektor Camera Position - Absolute World Position
2. Dále je hloubkový vektor transformován z View do World

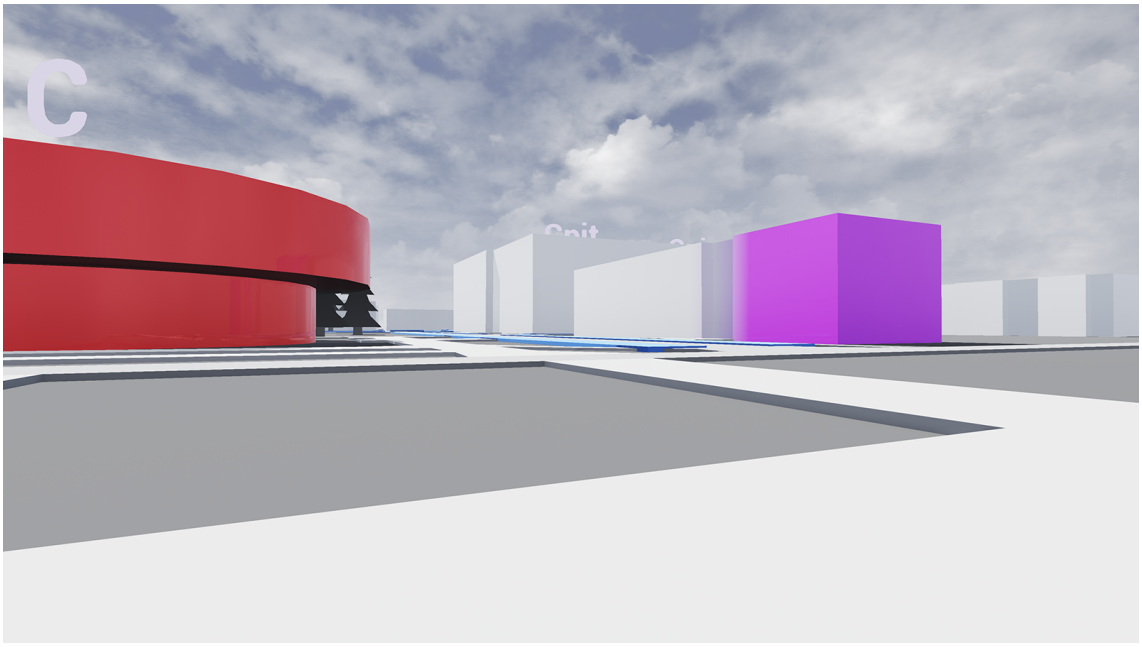


Obrázek 9: Ukázka využití parametru shaderu, vlevo hodnota přechodu barvy 10, vpravo 25

3. Následuje skalární součin mezi směrovým vektorem získaným v prvním kroku a transformovaným hloubkovým vektorem
4. Je zjištěna délka vektoru vypočítaného v prvním kroku, výsledek je vynásoben se skalárním součinem
5. Tímto je určena pozice kamery vzhledem k světu v rámci hloubky
6. Nyní je využita informace z Depth Bufferu upravená na základě vstupních parametrů, jenž určí vzdálenost a ostrost výsledného přechodu
7. Odečtením takto upravené hodnoty od hloubky pohledu samotného získáme skalární hodnotu určující hodnotu přechodu na základě vzdálenosti a vstupních parametrech přechodu.

Na základně výstupu z této funkce je provedena lineární interpolace mezi kódovou barvou budovy a barvou budovy, výsledek lze vidět na obrázku 10 .

Další funkcí je plošné zobrazení barevného kódování na všech budovách bez ohledu na vzdálenost od kamery. Zde byla využita absolutní pozice ve světě, jenž je vynásobená konstantou, čímž je určena ostrost přechodu. Následně je přičten dynamický parametr, čímž je určena výška přechodu. Poté je z absolutní pozice pomocí maskování vybrán pouze třetí koordinát, tedy Z, respektive B. Nakonec je daný kanál omezen pomocí funkce Clamp mezi 0 a 1. Výsledek je použit jako maska lineární interpolace mezi základní barvou a barvou konkrétní budovy v případě viditelného barevného kódování.



Obrázek 10: Ukázka využití Depth Fade pro obarvení budov v závislosti na vzdálenosti

[1]

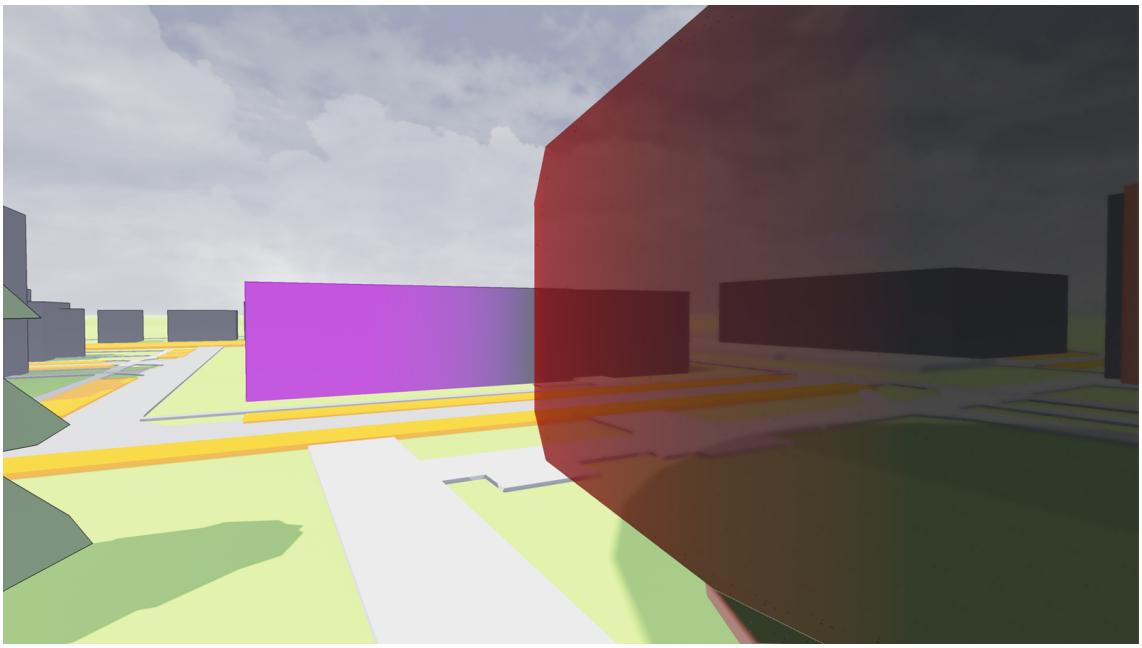
Materiál umožňuje dynamické snížení budov pro zlepšení viditelnosti značení budov pomocí písmen při pohledu z vlastních očí. Je využito skalárního dynamického vstupního parametru, jenž násobí absolutní Z pozici vertexu. Pochopitelně se jedná o operaci ve vertex shaderu a výstup tohoto shaderu je World Position Offset popsán v podkapitole 3.4.1.

Další skalární parametr ovlivňuje roughness jenž je blíže popsán v kapitole 3.2.

Součástí konceptu bylo zprůhledňování materiálu při přiblížení. Bohužel vzhledem k deferred renderování průhledné materiály mají velký počet omezení a není vhodné je používat jako materiály částečně neprůhledné. Nabízí se zde experimentální řešení v podobě funkce DitherTemporalAA. Jedná se o metodu, při které se využije náhodně generovaný vysokofrekvenční šum, čímž vytvoří úplnou průhlednost na náhodných pixelech. Postprocess antialiasing následně povrch zahladí a vytvoří tak iluzi částečně průhled-

[1] Dynamický parametr může být pouze typu Vector4, který je vždy čtyřkanálový. Tento vstup je v tomto materiálu často používán v lineární interpolaci s barvou v druhém kanálu, aby bylo možno provést lineární interpolaci, je nutno z dynamického parametru odstranit čtvrtý kanál, například za pomocí funkce BreakOutFloat4Components a následné MakeFloat3.

ného materiálu. Výsledek si můžete prohlédnout na obrázku 11. Tuto funkci je možno zapnout skrz uživatelské rozhraní za běhu virtuální prohlídky. Krom viditelných artefaktů je dalším omezením téměř nulová kontrola nad mírou průhlednosti a nutnost používat TemporalAA.



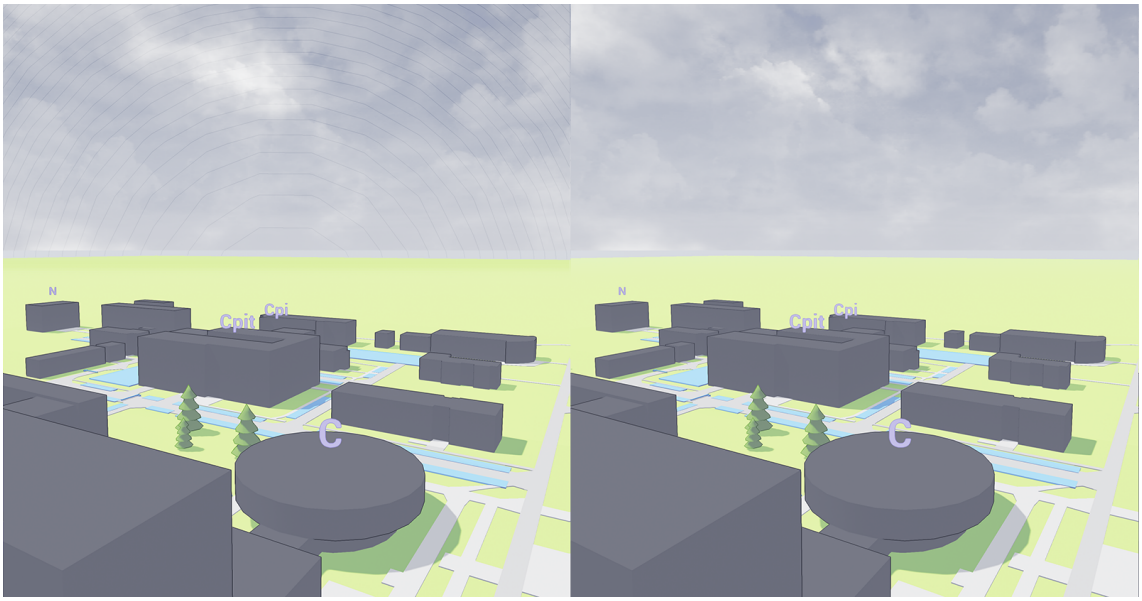
Obrázek 11: Ukázka využití Depth Fade pro obarvení budov v závislosti na vzdálenosti

3.4.5 Postprocessing materiál

Jeden z materiálu byl navržen pro použití v rámci postprocessu. Využívá tedy vyrenderovaných bufferů a na jejich základě provádí operace.

Konkrétním cílem toho shaderu bylo dosáhnout "kreslený"vzhled scény. Využívá detekci hran v depth bufferu. Hrany jsou dále testovány na hloubku vůči Depth bufferu, velmi vzdálené hrany jsou odříznuty, bylo zabráněno artefaktům na sky dome. Takto získané hrany jsou použity jako maska mezi dále předzpracovanou vstupní scénou a barvou hran.

Scéna samotná je prolnutá a přednastavená barvou pro podtrhnutí stylizace a sladění barev tvořící pastelový efekt.



Obrázek 12: Ukázka artefaktů vzniklých tvořením hran na sky dome

3.4.6 Materiál stopy

Pěší pawn byl navržen tak, ať za sebou zanechává stopu. Jedná se o barevný pruh, který zůstává po určitou dobu po průchodu pawna. Cesta se skládá z meshů, obdobně jako značení cest blíže popsaných v podkapitole 5.2.2.

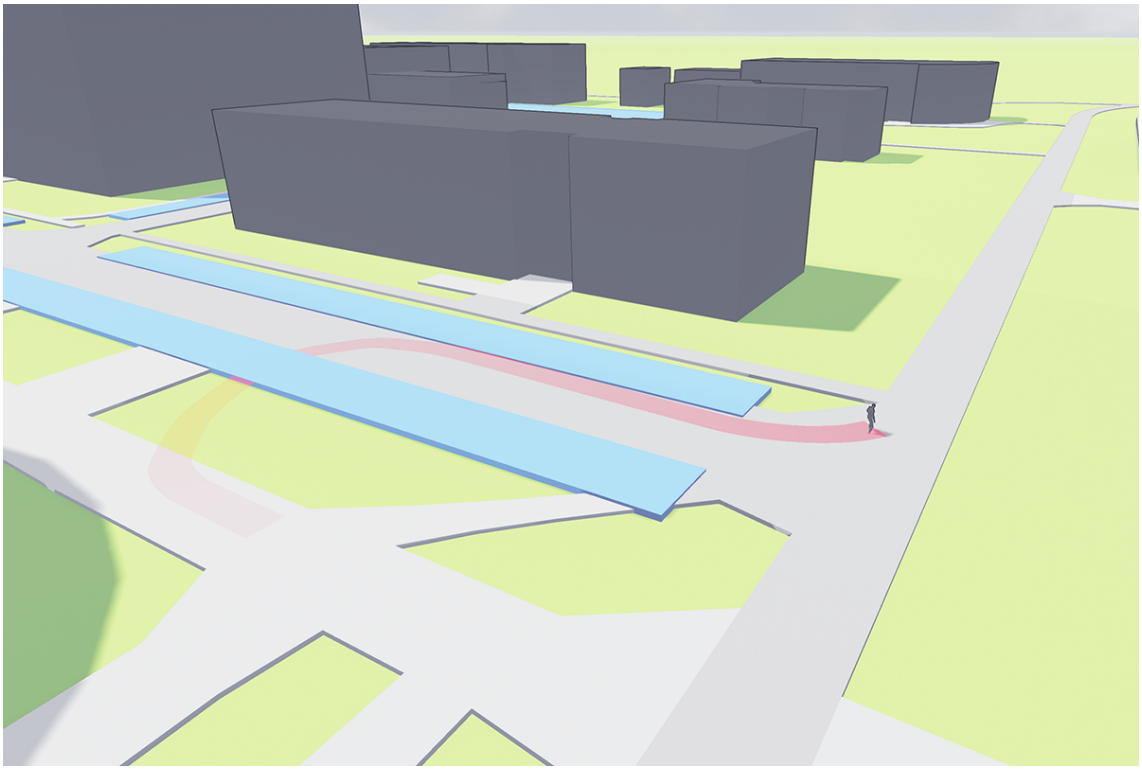
Každý segment v tomto případě mizí postupně skrze postupně se snižující viditelnost v závislosti na uplynulém čase. Tento efekt je řešen na GPU, při vytvoření segmentu cestu je do každého vertexu uložen údaj o času jeho vytvoření. Pří renderování je danému shaderu zodpovědnému za tento efekt poslán aktuální čas. Následně je spočítán normalizovaný čas:

$$age = CurrentTime - InitialTime \tag{6}$$

$$normalizedAge = \frac{age}{Duration} \tag{7}$$

Duration je doba, po kterou má být vytvořený segment viditelný. Pomocí normalizovaného času je poté spočítána viditelnost, v případě lineární změny viditelnosti:

$$opacity = lerp(1.0, 0.0, normalizedAge); \tag{8}$$



Obrázek 13: Ukázka stopy

Opacity je výsledná viditelnost v daný čas, lerp je univerzální funkce jak v HLSL, tak GLSL pro lineární interpolaci mezi dvěma hodnotami. Dříve bylo zmíněno, že v rámci renderovací pipeline jsou možnosti průhlednosti omezené. Tento materiál se renderuje v jiném průchodu než materiály aplikované na základní objekty scény, po dokončení výpočtu osvětlení a nemůže přijmat světlo, což umožňuje počítání průhlednosti bez zvětšení počtu průchodů renderu.

Materiál samotný obsahuje možnost přednastavit barvu stopy a dále dynamický parametr, který umožňuje vypnutí viditelnosti stopy skrze uživatelské rozhraní.

4 Vytváření assets

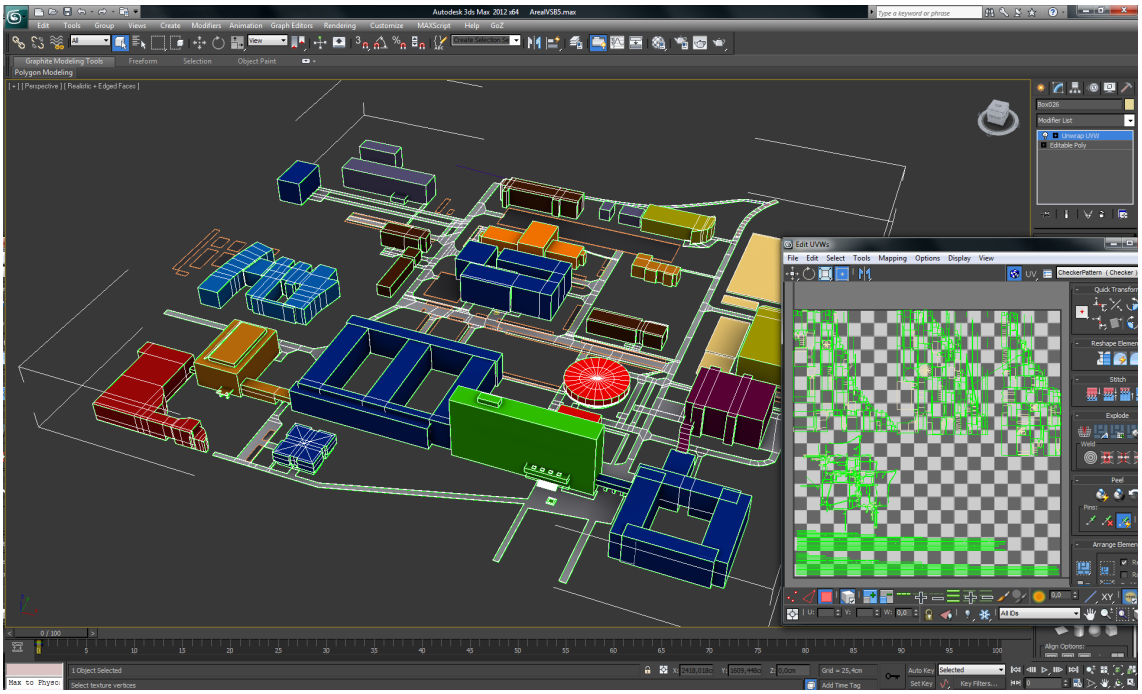
V této kapitole následuje shrnutí informací o vytváření modelů vizualizovaného areálu. Dále pak informace o importování vytvořených modelů, textur a prvků uživatelské rozhraní do enginu samotného.

Pro vytvoření potřebné 3d grafiky byl použit 3D Studio Max 2012. Díky cílové stylizaci bylo vytváření assets samotných snadné a jelikož se jedná o vedlejší část této práce, zmíním pouze stručně základní principy.

Vzhledem k použití v realtime enginu bylo nutné zachovávat čistou topologii meshe a optimalizovat UV mapování. Přesto, že ve finální verzi nejsou použity textury, UV koordináty jsou využity k zapečení světla a statických stínů do finální scény, jakékoliv překrývání je tedy nepřípustné a způsobilo by lehce viditelné artefakty.

K odlišení materiálu od sebe v rámci jednoho objektu bylo využito Material Id, který následně v UE umožnil aplikovat několik materiálů.

Během tvorby byly sjednoceny objekty pro cesty, chodníky a budovy, což umožnilo velmi rychlé iterace.



Obrázek 14: Ukázka vytváření optimalizovaného modelu s UVW mapováním pro virtuální prohlídku

4.1 Importování do UE

Pro importování 3D objektů byl využit formát .fbx, jenž je vyvíjen společností Autodesk. Jedná se o moderní, široce rozšířený binární souborový formát pro přenášení 3D modelů. Včetně informací o vertexech, normálách a UVW souřadnic umožňuje přenášet skeletal mesh, tedy informací o kostech objektu sloužících k animaci, což bylo v tomto projektu využito při tvorbě pěšího Pawna5.2.1 ovládaného uživatelem.

Jako formát textur využitých například v uživatelském rozhraní byl PNG. Jedná se o bezztrátový 32 bitový formát využívající 8 bitů na barevný kanál. Zde je nutno pamatovat, že se jedná o formát před importem, Unreal Engine samotný v rámci svého content manageru konvertuje textury dle požadavků zadaných při importu. V případě uživatelského rozhraní bylo využito bezztrátové komprese a potlačení vytváření automatické úrovně detailu vzhledem k jejich relativně neměnné velikosti. Pro vytváření mipmap 4.1 bylo využito bilinearního algoritmu zvětšujícího ostrost.

Definice 4.1 *Mipmapy jsou předem vypočítané a uložené optimalizované sekvence textur, které jsou použity zároveň s texturou původní. Výška a šířka každé následující textury je poloviční oproti předchozí. Slouží ke zvýšení rychlosti renderování a k potlačení aliasingu. Mipmapy je možno vytvořit pouze z textur, jejichž výška a šířka jsou mocniny dvou.*

5 Blueprinty - Vizuální programování

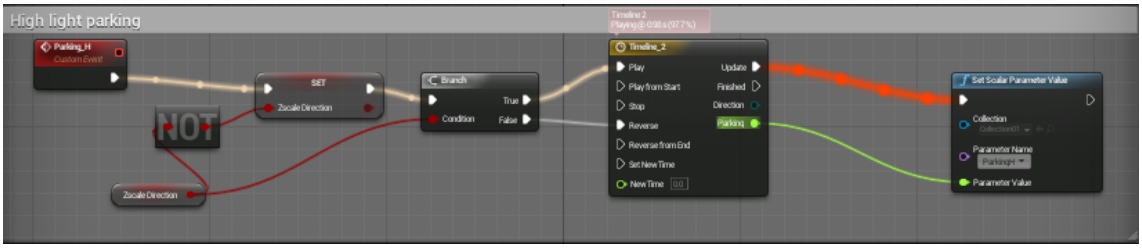
V této kapitole následuje popis použité metody implementace funkcionality za využití moderního prostředku v podobě vizuálního programování, jenž Unreal Engine nabízí.

Jedná se o scriptovací systém založený na uzlech. Jedná se o velmi flexibilní nástroj, který umožňuje vytvářet mechanismy a funkcionalitu.

Jednotlivé uzly reprezentují vše od matematických funkcí, přes marka až po celé třídy napsané v C++, které jsou ať už součástí zdrojového kódu, či napsány vývojářem.

Blueprint můžeme chápat jako třídu, jejíž instance jsou později vytvářeny ve hře dle potřeby. Mohou definovat či ovlivňovat ovládání hráče, herního světa, objektů, umělou inteligenci či uživatelské rozhraní.

Jednou z velkých výhod blueprintu je vizualizace datových toků, což je velmi dobře využitelné při ladění projektu a odstraňování chyb. Jak můžete vidět, na obrázku 15 je vizualizace konkrétní instance třídy za běhu programu. Ladění blueprintu umožňuje stejně jak ladění kódu vytvářet breakpointy, či procházet kód řádek po řádku, v tomto případě node po nodu.



Obrázek 15: Ukázka vizualizace datového toku v rámci blueprintu

Rovněž výsledky větvení a cykly jsou velmi dobře zobrazeny a umožňují snadnou kontrolu datových toků.

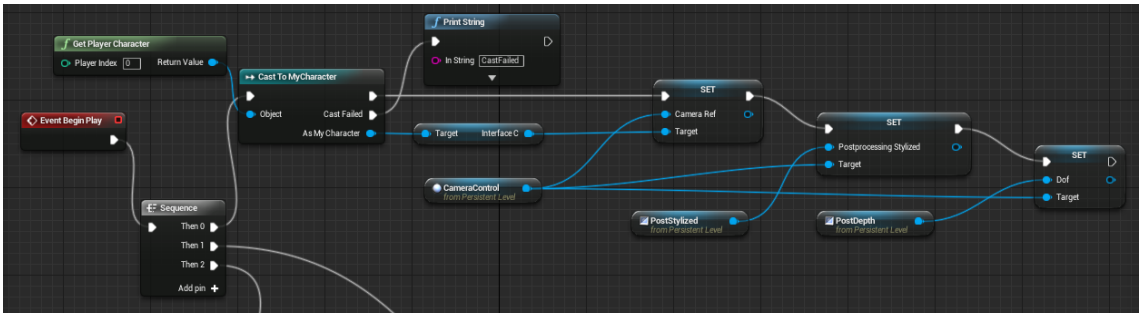
Problémem u komplexních projektů může být přehlednost či chybějící funkcionalita. V obou případech je řešení v rozšíření stávajících nodu či přidání vlastních skrz programování v jazyce C++. Tento úkon vyžaduje použití source verze Unreal Enginu, kterou je pochopitelně třeba zkompilovat. Unreal Engine od verze 4.5 umožňuje takzvaný Hot reload, tedy kompilovat zdrojový kód za běhu editoru a plynule tím propojit vývoj v blueprintu a C++ programování.

5.1 Komunikace mezi blueprinty

Pro předávání dat mezi blueprinty slouží takzváné Custom events, neboli vlastní události definované vývojářem. Aby bylo možno vyvolat Custom event jednoho blueprintu, v druhém musí daný blueprint znát referenci na instanci požadovaného blueprintu, k jehož eventu, události, chce přistoupit.

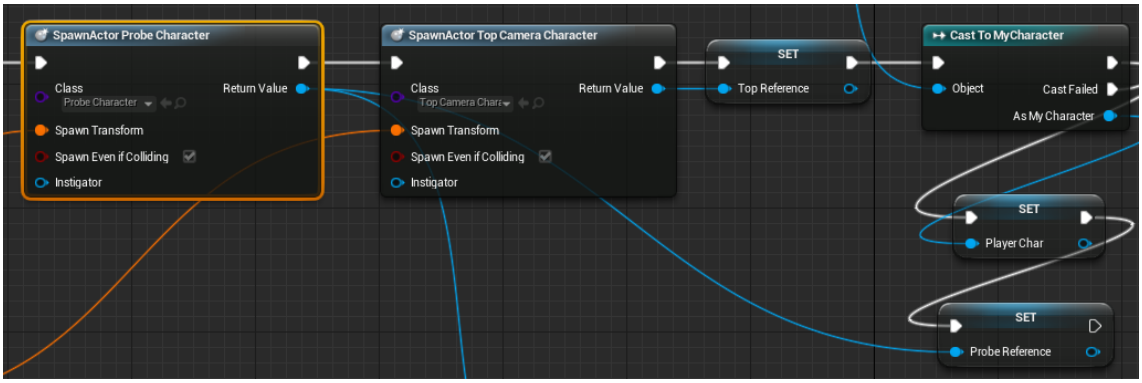
Získat referenci je možno několika způsobu, často se k tomu využívá takzvaný level blueprint. Jedná se o vždy existující blueprint aktuální herní plochy. Je zde možno přistupovat k referencím všech objektů v herním světě, tedy od statických meshů až po instance blueprintů.

Pro funkcionalitu tohoto projektu bylo nutno vytvořit několik takovýchto propojení. Předávání reference přes level blueprint bylo využito pro předání referencí požadovaného typu postprocesu třídě PresenationControll, dále pak propojení této třídy na uživatelské rozhraní umožňující ovládání přes grafický interface. Funkce pro předání referencí je zavolána jednou za využití Event Begin Play, tedy událost vyvolána při spuštění virtuální prohlídky.



Obrázek 16: Ukázka předání referencí přes level blueprint

Další možností získání reference na daný objekt či blueprint se nabízí, pokud blueprint daný objekt dynamicky vytváří. Této varianty bylo využito v Controlleru, jež umožňuje přepínání mezi různými způsoby pohybu v areálu, jak je blíže popsáno v podkapitole 5.2.1. Controller samotný vytváří instance Pawnu, mezi nimiž je přepínáno za běhu virtuální prohlídky, během jejich vytvoření dochází zároveň k uložení referencí, které jsou k této funkcionalitě využity.



Obrázek 17: Ukázka získání referencí při tvoření instancí

5.2 Implementace interaktivity

Hlavním důvodem volby herního enginu pro vytvoření virtuální prohlídky bylo poskytnout uživateli možnost projít daný areál dle svého uvážení, zvolit nastavení vizualizace a zvýraznit významné objekty dle potřeby. Unreal Engine implementaci značně ulehčil, přesto pochopitelně jeho základní funkcionalita nepočítá s některými možnostmi, které byly požadovány v tomto případě.

Vzhledem k nasazení na mobilní zařízení jako jsou tablety, bylo nutností zpřístupnit základní uživatelské funkce skrze grafické rozhraní tak, aby nebyla nutnost používat těžce přístupné klávesové zkratky jak je dále popsáno v podkapitole zabývající se grafickým rozhraním 6.2.

5.2.1 Pohyb

Pro vytvoření architektury ovládání pohybu byla využita základní struktura ovládacích prvků Unreal Enginu, která byla dále modifikována. Struktura se skládá z následujících částí:

- **Player** - slouží k jednoznačné identifikaci daného hráče, tedy uživatele.
- **Controller** - actor, jenž je přidělen hráči a který ovládá Pawna či jiného actora, který je odvozen z třídy Charakteru. Úzce komunikuje s pawnem, kterého ovládá a navzájem mezi sebou synchronizují rotaci, přesněji rotace pawna je většinou řízena rotací controlleru, která je odvozena z požadavku pawna, tedy například důsledku kolizí. Přebírá input, ať již z klávesnice či jiného ovladače jako například game-

padu, což je relevantní pro tento projekt, jelikož ovládání skrz tablet využívá twin stick principu. Potom, co je input přebrán ovladačem, je poslán pawnu, který se dále rozhodne dle svých atributů, jak s daným vstupem naložit.

Controller běžně ovládá jednoho pawna, v jistých případech může controller ovládat několik entit, či v konkrétním případě toho projektu umožňuje přepínání mezi různými druhy pawnu.

- **Pawn** - třída, z níž jsou odvozeny actors, které ovládá hráč či AI a tvoří reprezentaci daného uživatele přímo v herním světě, často se jedná přímo o herní charakter. Základní třída umožňuje definovat a limitovat velké množství atributů pohybu, jakožto rychlost chůze, výška výskoku, akcelerace, či rychlost rotace. Stejně tak umožňuje omezit osy pohybu. Skrze blueprinty či modifikaci zdrojového kódu je možno dále upravovat chování například pro následování vyznačených cest.

V rámci této struktury je běžné, že se každý prvek v rámci jednoho hráče vyskytuje právě jednou, neboli jeden hráč má přidělen jeden Controller, jenž má přidělen jednoho pawna. Tato struktura byla pro potřeby této virtuální prohlídky nedostatečná.

Ve finální verzi jsou Controlleru přiděleni tři pawni, tak aby nabídli komplexní, ale intuitivní prohlížení areálu.

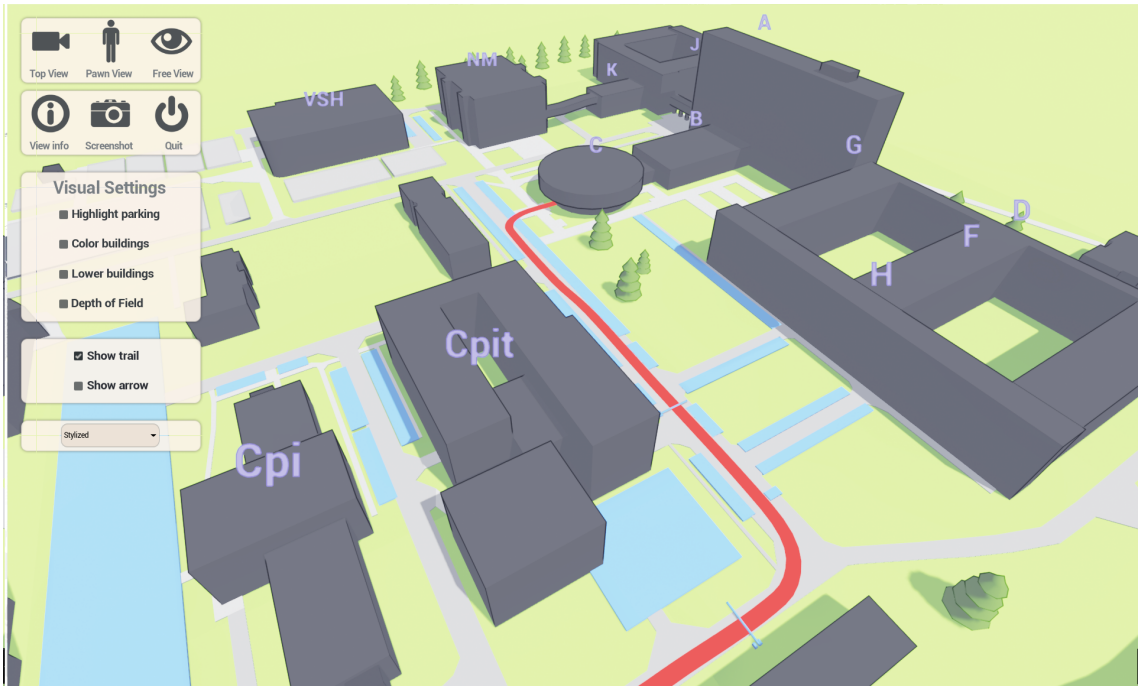
První možností je pěší Pawn, simulující procházení areálu z pohledu první osoby, tedy člověka včetně vizualizace postavy a reálné výšky očí.

Druhou možností je volný pohyb, jedná se o Pawna, který se dá přirovnat k létající sondě. Zde byla využita základní fyzika, akcelerace objektu a decelerace působením vzduchu pro simulaci hladkého pohybu. Dále bylo implementováno ovládání uhlu pohledu pomocí pohledu myší. Vzhledem k předpokládanému požadavku ze strany uživatele přistupovat k uživatelskému rozhraní existuje možnost dočasně vypnout pawna ovládání myši, tato funkce byla přiřazena klávesové zkratce a není zahrnuta v grafickém uživatelském rozhraní jelikož není relevantní pro ovládání na mobilních zařízeních. Poslední možností je kolmý pohled z vrchu s možností rotací kamery v ose Z.

5.2.2 Kreslení cest

Jednou z funkcí této virtuální prohlídky je kreslení cest uživatelem v prostoru. Tyto cesty je následně možno zachytit na screenshotu a usnadnit navigaci návštěvníkům areálu.

Funkcionalita byla realizována následovně:



Obrázek 18: Cesta vložená uživatelem do virtuální prohlídky

1. Při stisku přednastavené klávesy, původně F, je vytvořena instance třídy Path a umístěna do počátku souřadnicového systému. Reference na tuto instanci je následně uložena.

2. Při stisknutí pravého tlačítka myši je vytvořen raycast ze směru pohledu uživatele na kurzor myši. Vzhledem k neznámé vzdálenosti, ze které je raycast použit, můžou nastat problémy s přesností, tedy detekcí zásahu objektu, z tohoto důvodu dochází k ověření raycastu, neúspěšný raycast je zahozen.

3. Po úspěšném raycastu jsou předány informace o zásahu instanci třídy Path, na níž ukazuje reference uložena v prvním bodě.

4. Po obdržení informací instance Path vytvoří bod spline na daném místě s pevně přidělenou výškou Z. Následně projde všechny existující spline body a na jejich základech vytvoří cestu z předem nastaveného meshe podél dané spline. Pokud již existovala vytvořená cesta, jsou její viditelné meshe odebrány tak, aby bylo zabráněno redundantní geometrii. Vzhledem ke struktuře navázání meshe na spline není možno pouze přidávat a upravovat předchozí stav meshe a je proto nutno je

odebrat a provést kompletní rekonstrukci, nicméně se jedná o relativně nenáročnou operaci, která probíhá pouze při vstupu ze strany uživatele.

5. V případě vytvoření nové instance path je předchozí instance odebrána. Přesto, že aktuální verze umožňuje mít pouze jednu aktivní instanci Path, reference jsou uloženy do pole, je tedy možné velmi lehce modifikovat projekt tak, aby umožnil několik nezávislých cest. Tato možnost byla ponechána v případě změny požadavku na funkcionalitu v budoucnu.

5.2.3 Průlet

Efektní implementovanou funkcí je průlet kamery vytvořenou cestou. Během průletu kamera následuje poslední uživatelem vytvořenou cestu, rotace kamery je nastavena dle směru cesty v každém jejím bodě, jedná se tedy o úplné následování cesty.

Realizace probíhá následovně:

1. Po spuštění následování cesty je hráč přesunut do volného pohledu
2. Dojde k přemístění na začátek křivky, které definuje tvar cesty
3. Je spuštěna časová funkce, přednastavená na 10 sekund, funkce vrací hodnotu od 0,0 do 1,0 v závislosti na uběhlém čase
4. V závislosti na výstupu časové funkce je vypočítána pozice. Tedy výstup časové funkce vynásoben délkou křivky, takto získaná hodnota je vstupem funkce *Get World Position at Distance Along Spline*, jenž vrací pozici v závislosti na vzdálenosti
5. Dle aktuální pozice na křivce je přemístěn pohled kamery
6. Aktuální pozice je dále použita k vypočtení směru křivky v daném bodě, jenž je převeden na rotaci a dle něj je nastavena hodnota controlleru, jenž ovládá rotaci kamery
7. Po dokončení průběhu časové funkce, tedy kdy její výstup dosáhne hodnoty 1,0, je ovládání uvolněno a navráceno zpět uživateli

5.2.4 Vyznačování objektů

Uživatel má možnost zobrazit vyznačení důležitých objektů, jedná se o barevné rozlišení budov a zvýraznění parkovacích míst. Obě tyto funkce pomáhají k vizuální komunikaci s uživatelem a předávání informací, pomocí níž se může uživatel orientovat19.

Barevné vyznačení budov probíhá dvěma způsoby, přičemž oba jsou interaktivní:

- Uživatel může zapnout a vypnout obarvení všech existujících budov. Tato funkce byla realizována vytvořením blueprintu PresentationControl, jenž v závislosti na vstupu ovládá parametrizovaný materiál popsaný v podkapitole materiál budov3.4.4.
- Dále je možnost zobrazit barevné vyznačení budov v závislosti na vzdálenosti kamery, tedy pozice uživatele a daného objektu.

Obdobným způsobem funguje vyznačení parkovacích míst, vstup uživatele je opět zpracován v blueprintu PresentationControl.

5.2.5 Volba stylizace

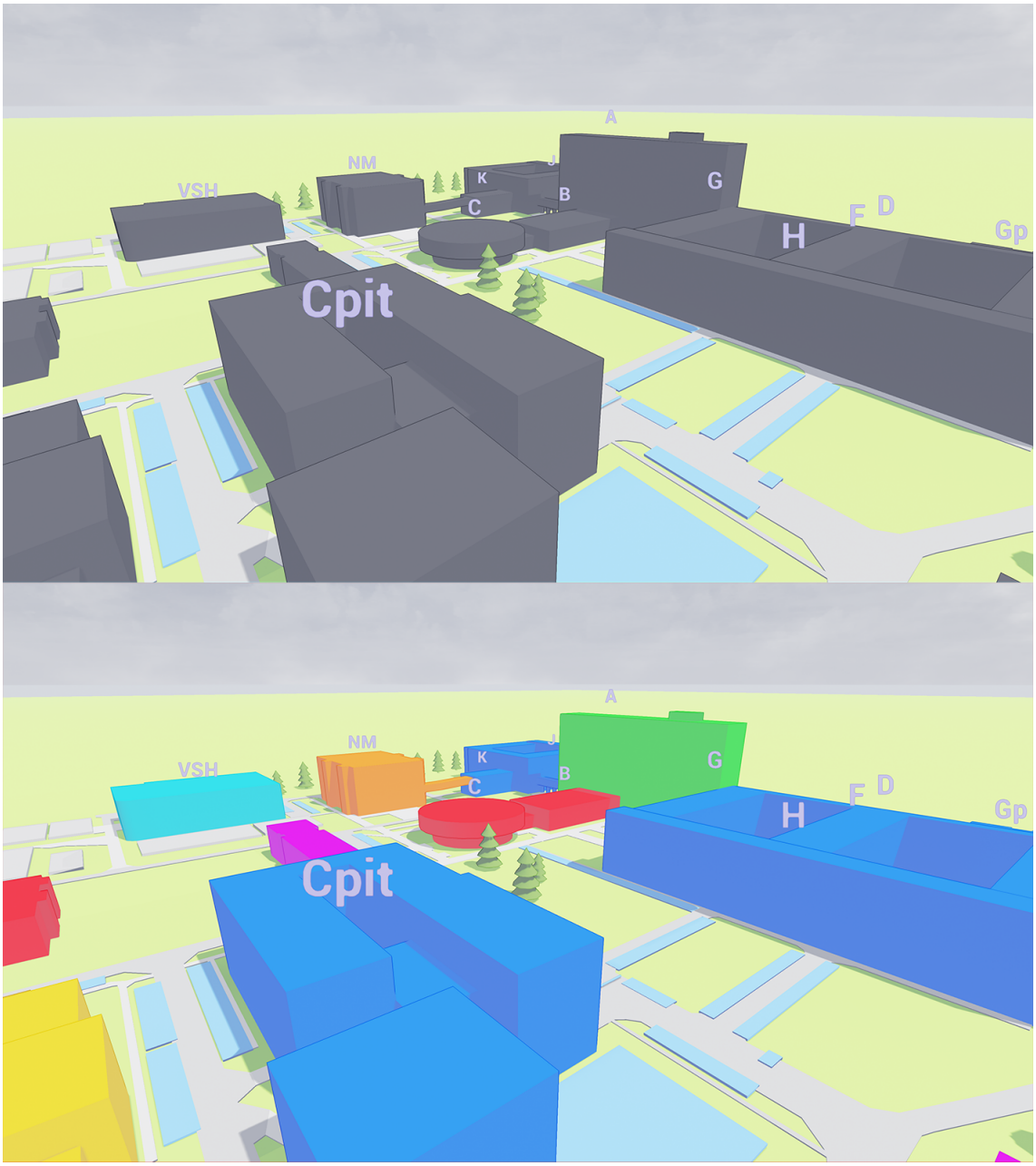
Významnou možností je volba jedné z několika přednastavených stylizací. Tato funkcionalita umožňuje zvolit nejvhodnější vizuální stylizaci pro daný účel prezentace.

V základní verzi projektu existují tři předem vytvořené stylizace. Přepínání mezi nimi probíhá skrz uživatelský interface a je opět řízeno pomocí blueprintu PresentationControl. Na základě přepínače je vybrána série přednastavených konstant, které jsou přiřazeny Material Parameter Collections, jenž ovládaní dané materiály, rovněž dochází k volbě odpovídajícího postprocessingu.

Systém byl postaven tak, aby bylo v budoucnu možné lehce přidat další stylizace, nicméně je třeba pamatovat, že hlavní výzvou zde zůstává správné použití a nastavení materiálu k dosažení požadovaných výsledků.

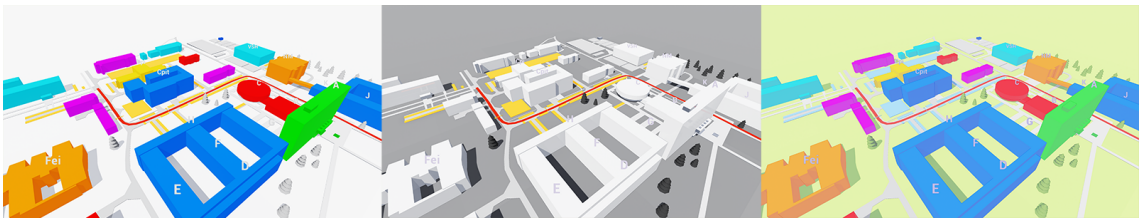
Dosáhnutí vzhledné stylizace není jednoduché. Přes značný vliv subjektivního dojmu je třeba akceptovat a aplikovat objektivní kvality. Jedná se zejména o harmonii barev a kontrast. V této vizualizaci, jenž nevyužívá textur, jsou hlavním původcem kontrastu stíny objektů. Ve stylizaci *cartoon* bylo dosaženo snížení kontrastu pomocí překrytí výsledného obrazu jednolitou barvou o vysoké průhlednosti.

Stylizace *high end* naopak nabízí vysoký kontrast. Cílem zde bylo dosáhnout moderního, technologicky pokročilého vzhledu. Stylizace využívá odrazu v prostoru obrazovky3.3.1.



Obrázek 19: Porovnání vizualizace s barevným značením budov a bez něj

Stylizace *basic* nabízí základní, ale přehledný vzhled za využití kontrastu tmavých budov na světlém podkladu. Z pohledu renderingu se jedná o nejjednodušší stylizaci.



Obrázek 20: Ukázka tří různých stylizací virtuální prohlídky

6 Uživatelské rozhraní

Uživatelské rozhraní slouží k ovládání aplikace uživatelem a lze rozdělit na dvě části. Samotné ovládání skrz příkazy z klávesnice či pohyb myši a dále grafické, tedy ovládání skrze ikony či jiné grafické prvky, zpravidla kliknutím myši.

6.1 Klávesové příkazy

Ovládání přes klávesnici je v UE řešeno pomocí událostí vzniklých při stisku klávesy. Všechny vstupy z klávesnice v rámci tohoto projektu jsou řešeny uvnitř vlastní definice třídy Controller, přesto je obecně možno využít i Level blueprint.

Vzhledem k požadavku o možnosti nasadit tento projekt na přenosné zařízení jako tablety, byla snaha omezit nutnost použití klávesnice na minimum.

V případě, že je projekt spuštěn na tabletu, je pro ovládání postavy použit dotykový joystick zobrazený na obrazovce tabletu.

Uživatelé na stolních počítačích, či zařízeních s klávesnicí, mohou stále využít zkratek zobrazených v nápovědě, jedná se například o přepínání mezi různými druhy pohledu, či tvoření nových vyznačení cest.

6.2 Grafické rozhraní

Požadavkem byl moderní interface, který nebude překážet vizualizaci samotné. Byl zvolen jednoduchý design, dále je možno grafické rozhraní skrýt. Grafické rozhraní není zahrnuto na screenshotech pořízených skrz program samotný.

GUI bylo rozděleno do čtyř oblastí:

- Ikony pro základní ovládání jako přepínání pohledů, pořízení screenshotu, zobrazení nápovědy či vypnutí aplikace
- Vizuální nastavení zobrazení, především přepínání funkcionality materiálu budov, zvýraznění parkování či obarvení budov
- Nastavení Pawna obsahující zobrazení směrové šipky a stopy
- Volba celkové stylizace vizualizace

Samotné GUI je inicializováno při vytvoření prvního, tedy pěšího Pawna, tato vlastnost se dá lehce změnit, jen je třeba ji brát v potaz v případě budoucí změny v projektu, konkrétně systému Pawnu.

Využito je klasických OnClickEvent, tedy událostí pří kliknutí na specifický ovládací prvek. Daná událost vyvolá vlastní událost vytvořenou v požadovaném blueprintu, vytvoření reference na instanci blueprintu je blíže popsána v kapitole 5.1.

V aktuální verzi aplikace je veškerá komunikace GUI jednosměrná, tedy volby v GUI ovlivňují vizualizaci, nikoliv naopak. Neobsahuje tedy žádnou HUD funkcionalitu, tedy změnu informací v závislosti na stavu aplikace. Vytvořená struktura GUI však tuto funkcionalitu umožňuje a je možné ji i implementovat jako budoucí rozšíření projektu.

7 Výkonnostní testy

Výkonnostní testy v této kapitole slouží k porovnání výkonu na různých zařízeních a k určení, zda je aplikace pro dané zařízení o daném výkonu vhodná. Pomáhá najít problematické části logiky aplikace a renderování, případně odhalit příliš komplikované materiály, či dokonce případnou nekompatibilitu funkcí s určitým zařízením.

7.1 Metodika testování

Pro účely porovnání výkonu byla vytvořena funkce Benchmark, využívá již implementovaného následování cesty popsané v kapitole 5.2.3. Po aktivaci kamery následuje 10 sekund dlouhý průlet přednastavenou křivkou. Na začátku je spuštěno ukládání údajů pomocí funkce StatFPSstart, po dokončení křivky přednastavené dráhy je ukládání ukončeno, na základě toho Engine vygeneruje csv tabulky s daty, jenž byly dále zpracovány a vyhodnoceny. Na každém zařízení byl test spuštěn třikrát, jednou pro každou z přednastavených stylizací. Stylizace se liší složitostí využitých shaderů a postprocessingů, byl tedy předpokládán rozdíl v celkové době renderování snímku.

Seznam zařízení, na kterých bylo provedeno výkonnostní testování je uveden v tabulce1.

Pro zachování perfektní plynulosti, tedy 60 snímků za sekundu, je třeba, aby bylo možno vyrenderovat snímek pod $16,7ms$. Za přijatelné se považuje snímková frekvence nad 30 snímků za sekundu, tedy do průměru $32,4ms$ na snímek.

Funkce StatFPS ukládá čtyři údaje pro každý vyrenderovaný snímek.

1. **Frame** - Celková doba vykreslení snímku.
2. **Game** (Game thread)- Vlákno logiky aplikace, tedy čistě CPU.
3. **Draw** (Render thread) - Vlákno renderovaní z pohledu CPU, drawcalls, nastavení parametrů, occlusion culling, nastavování a čištění bufferů.
4. **GPU** - Čas, jenž potřebovalo GPU k vykreslení daného snímku

7.2 Výsledky testování

Na grafech obrázku 21 je možno vidět výsledky pro testování na hardwaru Desktop 1. Snímková frekvence je velmi stabilní pro veškeré testované stylizace. Ve všech případech

Název zařízení	CPU	RAM	GPU	VRAM	Rozlišení
Desktop 1	i7-4770k	16 GB	EVGA GTX 770	4 GB	1920x1200
Desktop 2	i5-2500k	8 GB	Radeon HD 6970	2 GB	1920x1200
Tablet Samsung	Quad, 2.3GHz	3 GB	Adreno 330	sdílená	2560x1600
Notebook Acer	i7-4702MQ	16 GB	Nvidia GTX 760M	2 GB	1600x900

Tabulka 1: Tabulka testovaného hardwaru

se pohybuje nad 120 snímků za sekundu, tedy průměrnou dobu potřebnou na vyrenderování snímku pod $9ms$. V oblasti kolem sedmé sekundy je možno pozorovat značné snížení času potřebného pro Rendering thread. V této oblasti kamera dočasně zamíří na oblohu a počet renderovaných objektů se prudce sníží. Přesto výkon zůstává stabilní, je to dáno zejména celkovou optimalizací modelu areálu a Render Thread samotný tedy není omezujícím faktorem.

Game thread, tedy logika hry si po celou dobu benchmarku vyžádala pouze velmi nízký potřebný čas a není z pohledu výkonu nijak problematická a v rámci desktopu nenabízí prostor pro optimalizaci.

Rychlost vykreslování snímků je tedy limitována schopností GPU. Přes rozdílnost postprocessu a materiálů jsou rozdíly napříč stylizacemi v rámci odchylky měření. Rychlost vykreslování je tedy ovlivněna zejména minimální nutnou funkčností deferred renderingu a opět nenabízí mnoho prostoru pro optimalizaci při zachování vizuální kvality.

Obrázek 22 nabízí grafy výkonnostního testování pro zařízení Dekstop 2. Na tomto zařízení byl problém s rychlostí Rendering thread. Čas je stabilně příliš nízký a je pravděpodobné, že není možno jej korektně měřit na daném zařízení v rámci použitého enginu. Výkonnost je však opět omezena GPU. Průměrná snímková frekvence byla 66,5 snímků za sekundu a splňuje podmínku pro absolutní plynulost.

Přes stabilně stejný celkový výkon napříč stylizacemi lze pozorovat rozdíl v časech Game thread. Vzhledem k tomu, že volba stylizace nijak neovlivňuje logiku hry, je tento výsledek neočekávaný. S největší pravděpodobností se jedná o důsledek automatického přetaktování CPU, jenž se snaží přizpůsobit aktuální zátěži. Rovněž v tomto případě Game thread nevytvářel žádné omezení výkonnosti a hlavním omezujícím faktorem byla GPU.

Viditelné špičky času potřebného pro vykreslování snímku jsou vzhledem k nízkému potřebného času očekávány a jedná se o vlastnost GPU. Vzhledem k jejich nízkému počtu

a z pohledu absolutního času malé odchylky nepředstavují viditelné snížení kvality a neporušují iluzi plynulého pohybu.

Výsledky testování na zařízení Tablet Samsung lze vidět na obrázku 23. Na tomto zařízení byl problém s měřením GPU, což je přičítáno sdílené paměti a celkově jednodušší architektůře GPU v mobilních zařízeních, v tomto případě Adreno 330.

Časy pro render thread jsou velmi nízké. Právě ten je často omezující pro výkon 3D aplikací na zařízeních tohoto typu. Těchto výsledků bylo dosaženo díky optimalizaci modelu a využitím instancí parametrizovaných materiálů, což vedlo ke snížení počtu drawcallu, jenž mají největší dopad na čas render threadu.

Game thread vyžadoval neočekávaně dlouhý čas, je pravděpodobné, že nezměřený GPU je započítán do Game thread, čemuž nasvědčují i celkové potřebné časy.

I přes vysoké rozlišení se snímková frekvence udržuje v oblasti 30 snímků za sekundu a splňuje podmínku pro iluzi plynulého pohybu a i na tomto zařízení je virtuální prohlídka vizuálně obstojná.

V grafu lze vidět častější výkyvy, avšak díky svojí relativně malé četnosti a hodnotám nemají viditelný negativní dopad, nezpůsobují viditelné zaseknutí na konkretním snímku.

Průběh testování na druhém mobilním zařízení, Notebook Acer, je k dispozici na obrázku 24. Průměrná snímková frekvence 45,02 snímků za sekundu je dostačující pro efekt plynulého pohybu.

Obdobně jak u zařízení Desktop 2 nebylo možno získat čas pro Render Thread, avšak vzhledem k optimalizaci je jisté, že není omezujícím faktorem.

Celková snímková frekvence je velmi stabilní. Viditelný je pouze drobný rozdíl v oblasti kolem jedné sekundy, kde je celkový počet renderovaných objektů nižší a v oblasti kolem deváté sekundy, kde se do záběru dostane největší část areálu v nadhledu kamery. Přesto je rozdíl v potřebném času na snímek mezi těmito oblasti pod 3 ms.

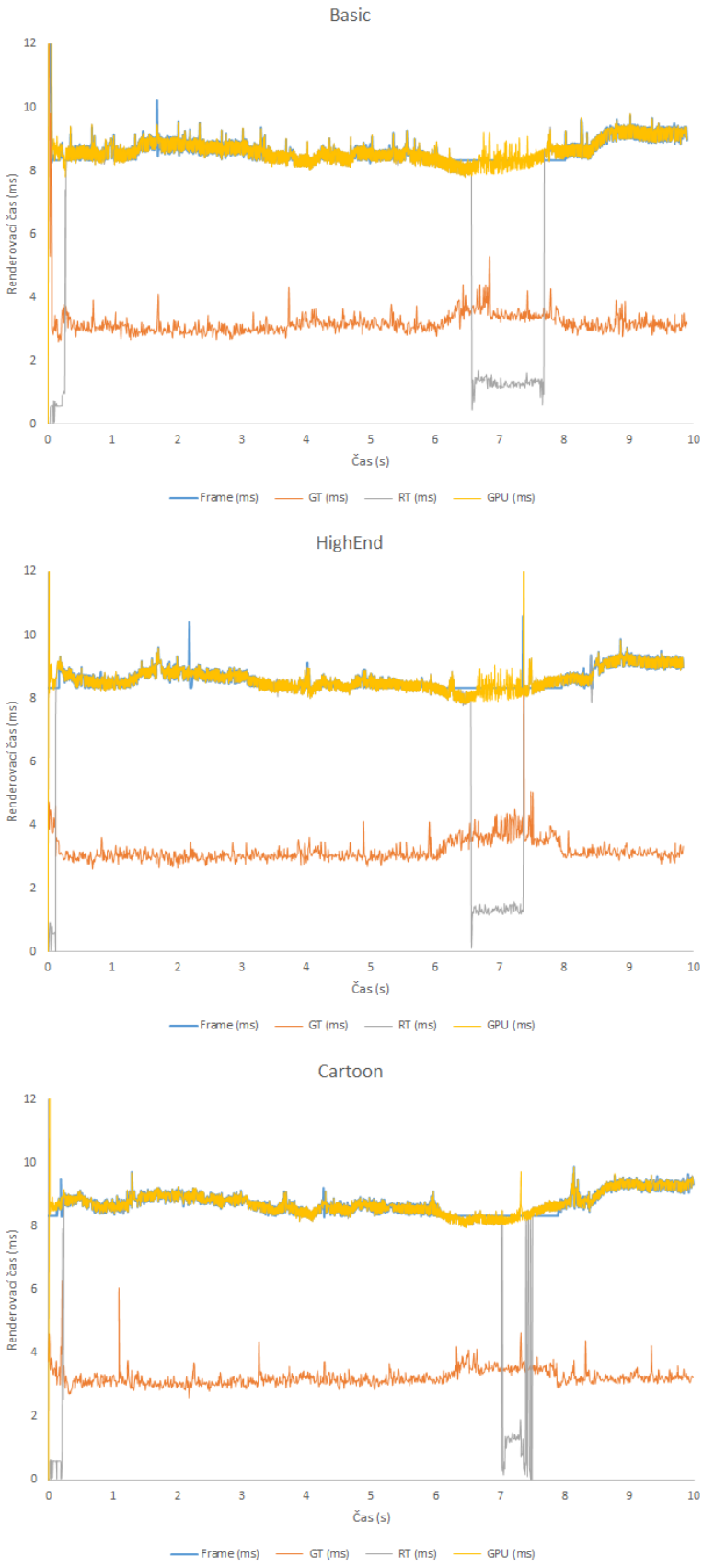
7.3 Shrnutí výsledků testování

Testy ukázaly, že provoz na moderních i lehce zastaralých desktopových zařízeních je bezproblémový a výkon vysoký. Rovněž pro moderní notebook není aplikace problém a nabízí stabilní výkon.

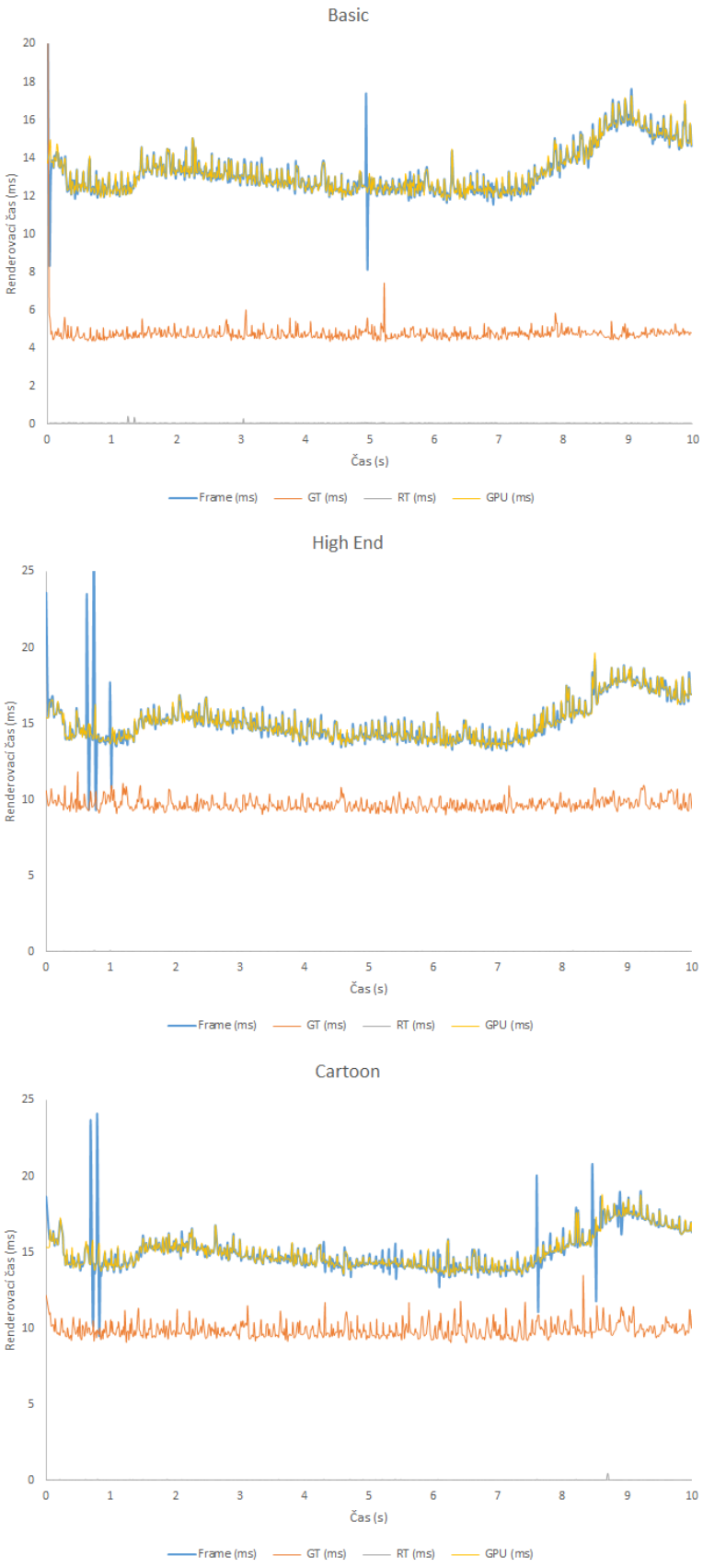
Na testovaném zařízení z kategorie tabletu byl výkon dostačující, avšak ne ideální. Pro vylepšení výkonu se nabízí snížení renderovacího rozlišení. Pro další optimalizaci

by bylo nutné snížení vizuální kvality omezením použitých materiálů a hlavně vypnutím postprocessingu, jenž je všeobecně pro zařízení typu tablet největší výzvou.

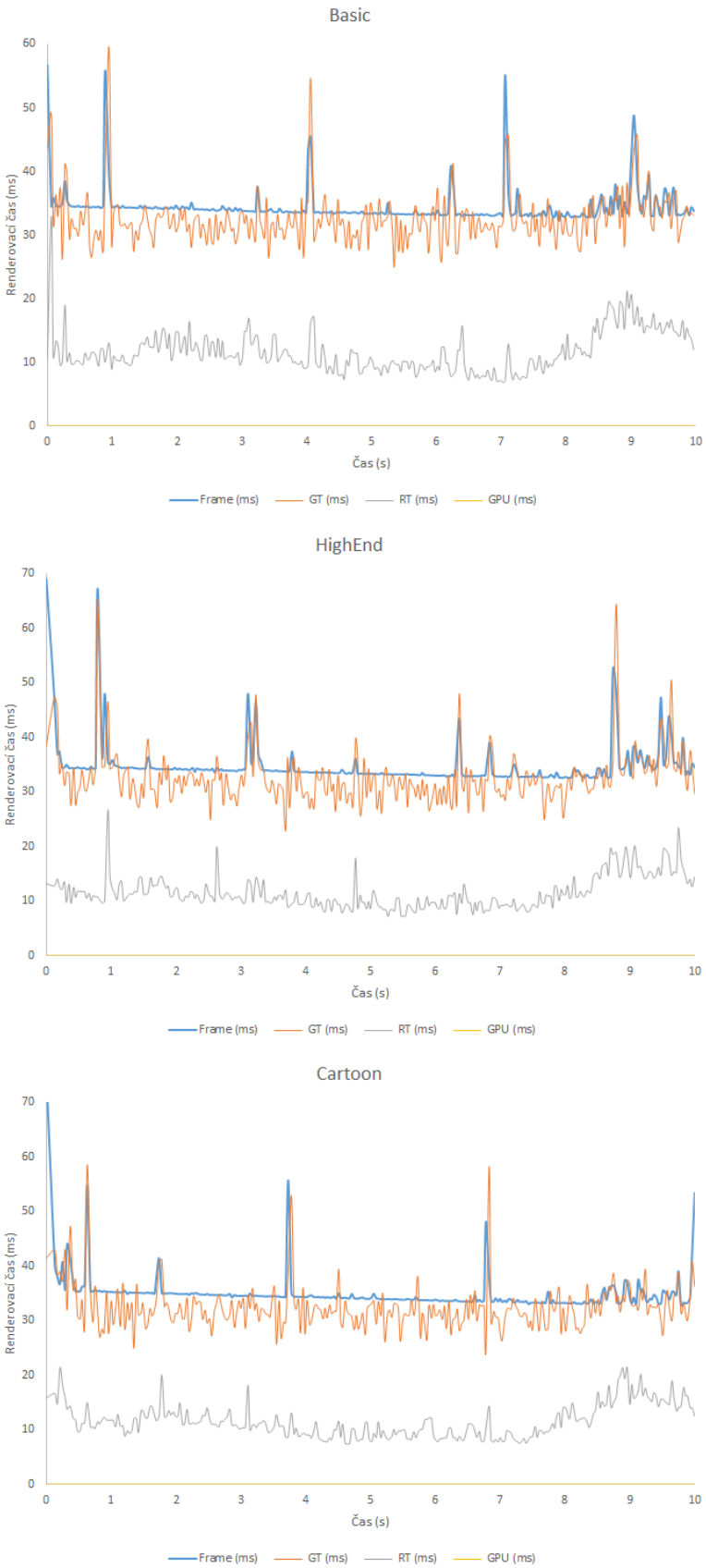
Logika hry, tedy Game Thread, je vysoce optimalizovaná a není problematická, stejně tak Render thread díky nízkému počtu objektů a unikátních materiálů. Výkon grafické karty má tedy největší vliv na plynulost této aplikace, což odpovídá očekávání v rámci použití herního enginu aktuální generace.



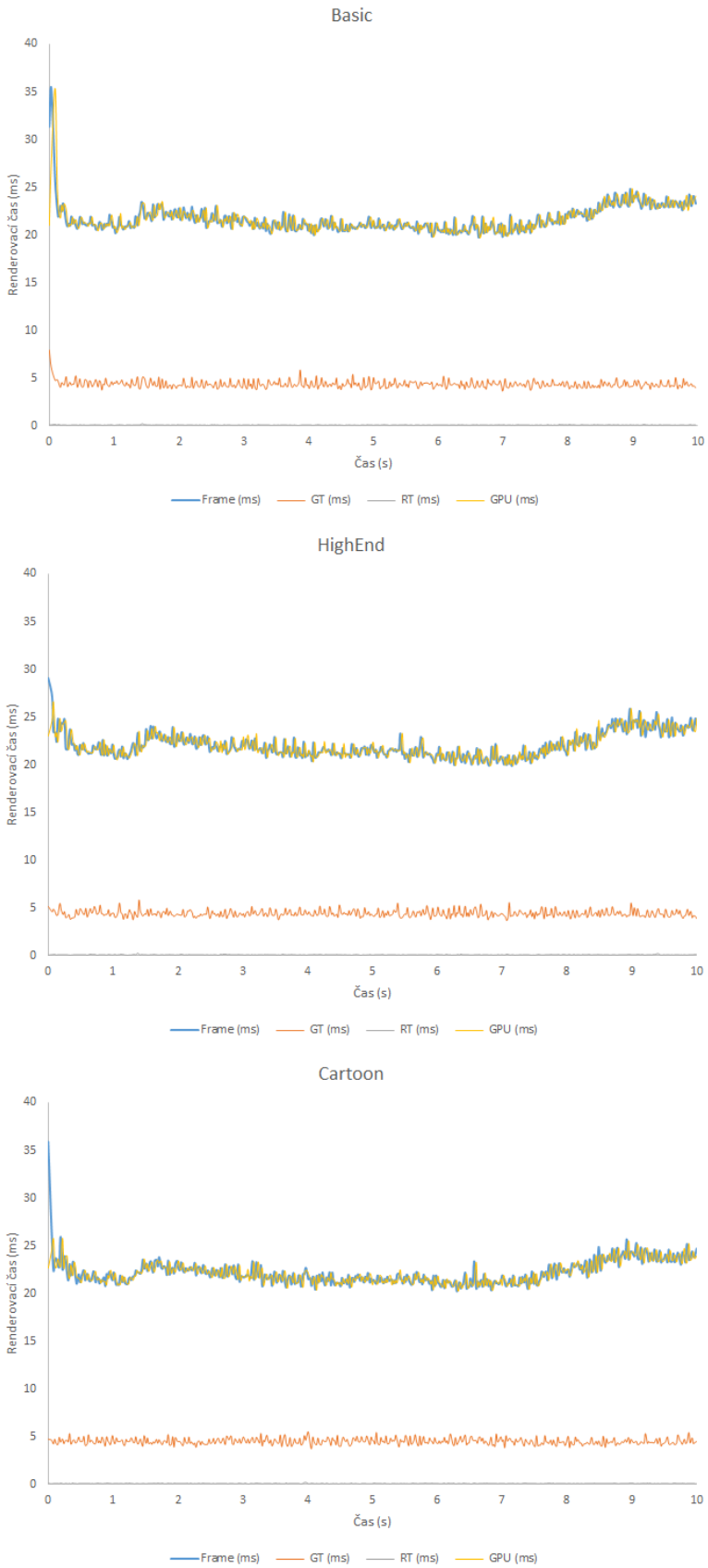
Obrázek 21: Výkonnostní test pro zařízení Desktop 1



Obrázek 22: Výkonnostní test pro zařízení Desktop 2



Obrázek 23: Výkonnostní test pro zařízení Tablet Samsung



Obrázek 24: Výkonnostní test pro zařízení Notebook Acer

8 Závěr

Hlavním cílem bylo vytvořit funkční aplikaci virtuální prohlídky areálu Vysoké školy báňské. Výsledkem je aplikace sloužící pro uživatele pro prohlídku areálu umožňující získat přehled o jeho areálu a usnadňující navigaci při reálné návštěvě. Aplikace je spustitelná a ovladatelná jak na desktopových počítačích tak výkonnějších mobilních zařízeních.

Pro dosažení tohoto cíle bylo vyhodnoceno několik alternativ nabízejících se pro implementaci dané aplikace. Na základě zkušeností a dostupných informací byla vybrána její nejvhodnější varianta.

Dalším úkolem bylo výkonnostní testování. Výkon byl testován na čtyřech zařízeních. Výsledky jsou vyhodnoceny a mohou posloužit jako orientační potřebný hardwarový výkon pro uživatelsky pohodlné ovládání této aplikace.

Unreal Engine se osvědčil jako výborný, flexibilní prostředek pro projekty požadující vysokou kvalitu vizuálního zpracovaní. Vzhledem k jednoduché stylizaci nejsou možnosti enginu testovány na samotnou hranu a nechávají prostor pro větší areály, či dokonce rozsáhlá území.

Text diplomové práce může posloužit jako nahlédnutí do oblastí rychle se vyvíjející moderní grafiky. Přes plánovanž příchod Directu 12, jenž řeší vykreslování velkých počtů objektů, zůstane mnoho věcí relevantních.

Demonstrace komplexnosti enginu však poukazuje, že pro některé projekty, hlavně graficky jednoduché, je stále vhodné takzvaně ušít engine na míru. Tedy za využití grafických knihoven, jenž nabízí například Ogre nebo Monogame vytvořit vlastní renderování a jádro optimalizované čistě na účely projektu.

Rád bych rovněž poukázal na důležitost uživatelského rozhraní. Během projektu byl ověřen fakt, že navrhovat uživatelské rozhraní a projekt částečně nezávisle a následně propojovat je nepraktický přístup. Obě části je v každém kroku nutno navrhovat jako funkční celek, v jiném případě mohou vzniknout omezení vynucující kompromisy, případně předělávání větších částí projektu.

Z pohledu uživatelského rozhraní je rovněž důležitá odezva uživatelů samotných. Připomínky testerů jsou velmi užitečný prostředek pro vylepšení ergonomie rozhraní, rozšíření či dokonce někdy omezení rozhraní, většinou z důvodu zjednodušení.

V neposlední řadě práce s vizuálním programováním, tedy blueprinty v prostředí UE, byla velmi rychlá a efektivní. Iterace bylo možno provádět rychleji než v případě

práce se samotným zdrojovým kódem a prototypování funkcionality přímo před koncovým uživatelem bylo velmi názorné, umožňující lepší vzájemné pochopení. Věřím, že tento přístup je velmi praktický pro rozsáhlé týmy, jenž mají rozdílně specializované členy jakožto grafiky a programátory a umožňuje lehčí pochopení funkcionality a omezení projektu při prezentaci členům jenž mají menší znalosti v oblasti programování.

Aplikace byla navržena jako robustní sytém, který je dále možno v rámci budoucího vývoje rozšiřovat. Nabízí se možnost vytvořit systém pro tvoření více cest, dále nechat uživatele zvolit barvu cesty za využití například HSL barevného modelu, nastavovat délku trvání průletu.

Jan Orszulik

9 Reference

[1] PHARR, Matt a Greg HUMPHREYS. *Physically based rendering: from theory to implementation*. 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2010, xxvii, 1167 p. ISBN 01-237-5079-2.

[2] OWENS, Brent. *Forward Rendering vs. Deferred Rendering.* [online]. 2013 [cit. 2015-04-25]. DOI: http://ruh.li/GraphicsCookTorrance.html. Dostupné z: http://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

[3] SOUSA, Tiago, Nick KASYAN a Nicolas SCHULZ. *Secrets of CryENGINE 3 Graphics Technology.* In: [online]. 2011 [cit. 2015-04-25]. Dostupné z: http://www.crytek.com/cryengine/presentations/secrets-of-cryengine-3-graphics-technology

[4] MCGUIRE, Morgan. *Screen Space Ray Tracing.* Casual Effects [online]. č. 2014 [cit. 2015-04-25]. Dostupné z: http://casual-effects.blogspot.cz/2014/08/screen-space-ray-tracing.html

[5] NGUYEN, Hubert. *GPU gems 3* [online]. Editor Hubert Nguyen. Upper Saddle River: Addison-Wesley, 2008, l, 942 s. [cit. 2015-04-25]. ISBN 978-0-321-51526-1. Dostupné z: http://http.developer.nvidia.com/GPUGems3/gpugems3_\part01.html

[6] WRIGHT, Daniel. *Deferred Shading in DirectX 11.* In: Unreal Development Network [online]. [cit. 2015-04-25]. Dostupné z: https://udn.epicgames.com/Three/DeferredShadingDX11.html

[7] EPIC GAMES, Inc. *Unreal Engine Documentation.*[online]. Epic Games, 2014 [cit. 2015-04-25]. Dostupné z: https://docs.unrealengine.com/

[8] EPIC GAMES, Inc. *Unreal Engine: What is GitHub* [online]. 2015 [cit. 2015-05-01]. Dostupné z: https://www.unrealengine.com/ue4-on-github

[9] EPIC GAMES, Inc. *Unreal answer hub* [online]. 2014. vyd. [cit. 2015-04-25]. Dostupné z: https://answers.unrealengine.com/

[10] TEAM. *Monogame* [online]. 2015 [cit. 2015-04-25]. Dostupné z: http://www.monogame.net/

[11] RUSSELL, Jeff. *Basic Theory of Physically-Based Rendering*. Marmoset Real-Time rendering [online]. [cit. 2015-04-25]. Dostupné z: https://www.marmoset.co/toolbag/learn/pbr-theory

[12] *The MIT License*. Open Source Initiative [online]. [cit. 2015-04-25]. Dostupné z: http://opensource.org/licenses/MIT

[13] RÁKOS, Daniel. *Efficient Gaussian blur with linear sampling* Rastergrid Blogsphere [online]. 2010 [cit. 2015-04-26]. Dostupné z: http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/

[14] *FireFox Nightly* [online]. 2015 [cit. 2015-04-28]. Dostupné z: https://nightly.mozilla.org/

[15] CAMP, Dave a Jason WEATHERSBY. *Firefox Developer Edition 38: 64-bits and more*. Mozilla Hacks. 2015. Dostupné z: https://hacks.mozilla.org/2015/03/firefox-developer-edition-38-64-bits-and-more/

[16] LOTTES, Timothy. *FXAA* [online]. 2009 [cit. 2015-04-29]. Dostupné z: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

[17] *Physically Based Rendering - Cook–Torrance*. ALAMIA, Marco. Coding Labs [online]. [cit. 2015-04-29]. Dostupné z: http://www.codinglabs.net/article_physically_based_rendering_cook_torrance.aspx

[18] UNITY TECHNOLOGIES.*Unity 3D* [online]. 2015 [cit. 2015-04-29]. Dostupné z: https://unity3d.com/

[19] EPIC GAMES. *Unreal Engine* [online]. 2015 [cit. 2015-04-29]. Dostupné z: https://www.unrealengine.com/what-is-unreal-engine-4

[20] CRYTEK. *CryEngine* [online]. 2015 [cit. 2015-04-29]. Dostupné z: http://cryengine.com/

[21] TORUS KNOT SOFTWARE LTD. *Ogre: Open Source 3D Graphic Engine* [online]. [cit. 2015-04-29]. Dostupné z: http://www.ogre3d.org/

[22] OKITA, Alex. *Learning C# programming with Unity 3D.* xix, 670 pages. ISBN 978-146-6586-529.

[23] MÖLLER, Tomas Akenine. 2008. *Real-time rendering. 3rd ed.* Wellesley, Mass.: A.K. Peters, xviii, 1027 s. ISBN 978-1-56881-424-7.

[24] VARCHOLIK, Paul. 2008. *Real-time 3D rendering with DirectX and HLSL: a practical guide to graphics programming. 3rd ed.* Wellesley, Mass.: A.K. Peters, xiv, 569 pages. ISBN 03-219-6272-9.

[25] ROST, Randi J. 2010. *OpenGL shading language: a practical guide to graphics programming. 3rd ed.* Upper Saddle River: Addison-Wesley, xliii, 743 s., [16] s. barev. obr. příl. ISBN 978-0-321-63763-5.

A Linearní Gaussovo rozostření

```
uniform float offset[MAX_SAMPLES]; //max samples settings depending on application
uniform float weight[MAX_SAMPLES];
int sampleCount;

float2 viewport;

sampler TextureSampler : register(s0)
{
    magfilter = LINEAR;
    minfilter = LINEAR;
    mipfilter = LINEAR;
    AddressU = clamp;
    AddressV = clamp;
};

struct VertexToPixel
{
    float4 Position : POSITION;
    float2 TexCoord : TEXCOORD0;
    float4 Color : COLOR0;
};

struct PixelToFrame
{
    float4 Color : COLOR0;
};

VertexToPixel FullVertexShader(float4 color : COLOR0, float2 texCoord : TEXCOORD0, float4
    position : POSITION0)
{
    VertexToPixel Output = (VertexToPixel)0;
    float4 positionL = position;
    position.xy -= 0.5; // half pixel offset for corret texel rendering

    // Vertex position mapping into screen space
    position.xy = position.xy / viewport;
    position.xy *= float2(2, -2.0);
    position.xy -= float2(1, -1);
```

```
    //Set vertex shader output structure
    Output.Position = position;
    Output.TexCoord = texCoord;
    Output.Color = color;

    return Output;
}

PixelToFrame HorizontalBlur(VertexToPixel PSIn) : COLOR0
{
    PixelToFrame Output = (PixelToFrame)0;


    float4 diffuseMap = tex2D(TextureSampler, float2(PSIn.TexCoord.x, PSIn.TexCoord.y)) *
        weight[0];

    float4 final;

    for (int i=1; i<sampleCount; i++) {
      diffuseMap +=
        tex2D(TextureSampler, ( PSIn.TexCoord+float2(offset[i], 0.0) /viewport.x ) )
          * weight[i];
      diffuseMap +=
        tex2D(TextureSampler, ( PSIn.TexCoord-float2(offset[i], 0.0) /viewport.x ) )
          * weight[i];
    }

    Output.Color = diffuseMap;
    return Output;
}

PixelToFrame VerticalBlur(VertexToPixel PSIn) : COLOR0
{
    PixelToFrame Output = (PixelToFrame)0;


    float4 diffuseMap = tex2D(TextureSampler, float2(PSIn.TexCoord.x, PSIn.TexCoord.y)) *
        weight[0];

    float4 final;
```

```
    for (int i=1; i<sampleCount; i++) {
        diffuseMap +=
            tex2D(TextureSampler, ( PSIn.TexCoord+float2(0.0, offset[i]) /viewport.y ) )
                * weight[i];
        diffuseMap +=
            tex2D(TextureSampler, ( PSIn.TexCoord-float2(0.0, offset[i]) /viewport.y ) )
                * weight[i];
    }

    Output.Color = diffuseMap ;

    return Output;

}
```

Výpis 2: Dvou průchodový shader pro gaussovo rozostření využívající lineárního samplování

```
    void CalculateLinearGauss()
    {

        float [] offsetsL = new float[(offsets.Length + 1) / 2];
        offsetsL[0] = offsets [0];
        float [] weightsL = new float[(weights.Length + 1) / 2];
        weightsL[0] = weights[0];
        for (int i = 1; i < weightsL.Length; i++)
        {
            weightsL[i] = (weights[(i * 2) - 1] + weights[i * 2]) ;
        }

        for (int i = 1; i < offsetsL.Length; i++)
        {
            offsetsL[i] = ((offsets [(i * 2) - 1] * weights[(i * 2) - 1] + offsets[i * 2] *
                weights[i * 2]) / weightsL[i]) ;
        }
        offsets = offsetsL;
        weights = weightsL;
    }
```

Výpis 3: přepočet offsetů a vah na lineární samplování